# Efficient Simulation of Message-Passing in Distributed-Memory Architectures

by

Erik Demaine

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 1996

# Abstract

In this thesis we propose a distributed-memory parallel-computer simulation system called *PUPPET* (Performance Under a Pseudo-Parallel EnvironmenT). It allows the evaluation of parallel programs run in a pseudo-parallel system, where a single processor is used to multitask the program's processes, as if they were run on the simulated system. This allows development of applications and teaching of parallel programming without the use of valuable supercomputing resources. We use a standard message-passing language, MPI, so that when desired (e.g., development is complete) the program can be run on a truly parallel system without any changes.

There are several features in PUPPET that do not exist in any other simulation system. Support for all deterministic MPI features is available, including collective and non-blocking communication. Multitasking (more processes than processors) can be simulated, allowing the evaluation of load-balancing schemes. PUPPET is very loosely coupled with the program, so that a program can be run once and then evaluated on many simulated systems with multiple process-to-processor mappings. Finally, we propose a new model of direct networks that ignores network traffic, greatly improving simulation speed and often not significantly affecting accuracy.

# Acknowledgements

First and foremost I would like to thank my advisor, David Taylor. He provided great support throughout my programme. I am overly impressed by how fast he reads my papers while catching the mistakes. I would like to thank the two readers, Thomas Kunz and Ken Salem, for their comments on this work. Thomas has also read some of my papers, to which he provided crucial comments, even on short notice. Ken Salem amazingly took on the role of a reader only one week before this thesis went on display.

A large part of this thesis was the experiments, which required parallel systems. The Shoshin group provided the network of IBM RS/6000 workstations. Thanks to the Dept. of Oceanography of Dalhousie University for access to Dalhousie's IBM SP2. Joe Potworka of IBM made several tools and contacts available, including the SP2 at Queen's University. Christian Foisy provided test programs and access to the SP2 at Université de Sherbrooke. The interest expressed by the above people made this work more enjoyable.

Without my friends here at the University of Waterloo, my programme would have been tremendously boring. I would like to thank in particular Research Associate Mike Nidd for his help in getting this acknowledgement section to work, among other things.

Most of all I would like to thank my father, Marty Demaine, for his continual love and support, particularly when I was frustrated gathering results. He also gave impressive comments for someone who knew little about the subject. My dog, The Missing Link, was particularly good at guarding my house while I worked. Although he didn't make any comments, I think he understands the world more than anyone.

Thanks go to the many bands and musicians for making great music that helped a lot during the writing phase, and in fact all my writing. Contemporary artists include REM, U2, The Cranberries, Timbuk 3, and The Tragically Hip; classical artists include Amadeus Mozart, Ludwig van Beethoven, Johann Sebastian Bach, and George Gershwin.

Finally, thanks to the Natural Sciences and Engineering Research Council (NSERC) and the Information Technology Research Centre (ITRC) for their financial support.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

A popular method for testing parallel programs, which we call *pseudo-parallelism*, is to use a single workstation and emulate a parallel computer via multitasking or multithreading. One can develop, debug, and test the correctness of a parallel application, without using valuable supercomputing resources. To use such resources at the development phase of a program is most often a waste. For example, it may not yet be clear that an iterative algorithm ever satisfies a convergence condition, causing an infinite loop; this may in turn cause problems with the supercomputer's scheduler. It is also useful to run a program in interactive mode for debugging purposes, which is not possible in several supercomputing setups. Once the program is ready to go, one can port the program to a desired parallel computer; this is usually a simple task provided a portable language is used.

Another important application of pseudo-parallelism is education. Educational institutes do not need to obtain expensive supercomputing resources to teach parallel programming. Students obviously do not require high performance to obtain experience in parallel programming, provided there exists a parallel environment on the pseudo-parallel system that has similarities to a “real” parallel computer.

A problem does arise, however. How do students know how efficient their programs are? For example, how much speedup would be achieved if the programs were run on a particular truly parallel system? This question does not only apply to students; researchers would greatly appreciate insight into the performance of their parallel algorithms without the use of supercomputing resources. In particular, this applies to performance debugging; since supercomputing power would not (yet) be fully exploited, and so should not likely be used.

The solution is to simulate the supercomputer, subject to the parallel program as the “workload.” At least conceptually, the processes are delayed at certain points of execution so that the processes think that they are running on a supercomputer, and that they are running in parallel. One can then measure a value such as the total parallel execution time, and hence derive speedup.

The area of parallel-computer simulation is new and a great amount of interest has been paid in recent years; previous work is detailed in Chapter 2. However, there is still a significant amount of work to be done. We shall focus on distributed-memory parallel computers, since they offer higher scalability and performance than shared-memory multiprocessors; in addition, less interest has been given to the simulation of distributed-memory systems. Some of the topics which we address in this thesis are ease-of-use, trading accuracy for efficiency, and the simulation of multitasking. We also see how simulation can be used in combination with performance-visualization tools (such as ParaGraph) and with the message-passing standard MPI.

## 1.1 MPI

*MPI* (Message-Passing Interface) [40] is becoming an industry-wide standard for message-passing. The MPI Forum consists of universities, research centers, national laboratories, and multi-national high-performance-computing vendors. MPI was designed by over forty organizations and sixty people. The initial version, MPI1 [29], was published in June, 1993. The latest MPI (1.1) [55] was published in late 1994. MPI-2 will be published this year at Supercomputing'96. The goal is to create a universal communication language implemented on all parallel systems. The MPI Forum studied the existing message-passing systems, and instead of taking one as the "standard," they took the most attractive features out of these systems. MPI is based on CHIMP, PICL, EUI (for IBM SPx), Zipcode, p4, CMMD (for CM5), TCGMSG, PVM, NX (for Intel NX/2), PARMACS, Express, and Chameleon [55]. The main advantage of MPI is portability; it has been implemented (both in public domain and commercially) on a variety of parallel systems and networks of workstations. Many free MPI implementations, such as LAM [13] and MPICH [10], support pseudo-parallelism.

Several libraries for profiling MPI programs have been developed [10]. Unfortunately, all of them rely heavily on a shared real-time clock and record events of a program execution with this value. When the MPI program is run in a pseudo-parallel manner, this does not provide data useful for predicting behavior in a truly parallel environment. First, since multiple processes must share the machine, each pseudo-processor effectively sees the machine at $1/n$th of its speed, where $n$ is the number of pseudo-processors (we do not take into account the operating-system overhead of context switches, making real-time values even worse). Thus, if the program is completely parallel, no speedup would be reported by a typical MPI profiling library. Second, communication is measured incorrectly: on a single uniprocessor workstation, no network is involved and the operating system converts "message passing" to shared-memory references, which are often much faster operations. Even on a parallel computer, using real-time often yields incorrect results: the latest supercomputers, as they rarely cost under US$10,000,000 [37], are often shared, and so both the CPU and network speed viewed by the profiling library are inaccurate.

## 1.2 PUPPET

As we have seen, on both pseudo-parallel and truly parallel computers, the existing MPI profiling libraries are insufficient when full parallelism is not available. The proposed system, called PUPPET, attempts to remedy this problem with a set of tools including an MPI profiling library that is based primarily on (user) CPU time and ignores communication time (since this is inaccurate without full parallelism). The profiling library (or *logging library*) is based on the assumption that a program consists of computation and communication, and ignores other factors such as I/O. The *simulator* (another tool provided) takes a program execution log along with information about a network, and estimates true parallel execution time and speedup based on this network. Finally, the *network evaluator* is a program that estimates the network speed assuming that the computer is dedicated to running the evaluator.

PUPPET is useful in several ways. It allows effective evaluation of parallel programs without the use of a parallel computer (that is, on a pseudo-parallel machine). Researchers can implement various parallel algorithms and compare their running times and speedups. When one technique is chosen, it can be run on a supercomputer without any change to the program's code. Students can implement and test parallel programs and can also compare them; no supercomputing resource at their university is required. In fact, a student can estimate program run-time on any distributed-memory supercomputer, testing different topologies and speeds of the network. Assuming sufficient memory is available on the workstation, as many (pseudo) processors as desired may be used, thereby testing scalability of algorithms without a massively parallel computer.

### 1.2.1 Features

A key difference between PUPPET and existing parallel-computer simulations is that PUPPET is designed for MPI. To use PUPPET does not involve learning a new (simulation) language, and possibly recoding existing algorithms in that language. In fact, one need only link in the logging library, run the program as usual, and then run the simulator to collect results. The simulator can be run many times on the same log with various simulation parameters. This allows several questions such as "what if I used the following hypercube multicomputer instead?" to be answered rapidly.

The main theoretical contribution of PUPPET is its approximate network model, which is what makes the simulation phase so fast. The idea is to ignore network traffic and congestion, and simply approximate the time to send a message by assuming an otherwise isolated system. For most applications, this does not significantly affect accuracy, but it has a much better performance than, say, a detailed network simulation.

PUPPET uses CPU time and a non-traffic network model. Sometimes it may be useful to obtain higher accuracy than what these features offer. Fortunately, PUPPET is a fairly flexible system. It is possible to use other computation-time measures, such as augmentation or instruction-level simulation, as well as a network simulator or a different analytical model. There is no point in developing such components in this project, since they have received significant attention before.

PUPPET supports all deterministic features of MPI. In addition to typical send/receive operations, non-blocking operations (including a wait operation to delay until an operation completes) are available. One can also use synchronization, multicasting, reduction, and other collective operations of MPI. As far as we know, these features are unique to PUPPET.

Several modern-day programs exploit concurrency in addition to parallelism. That is, they use more processes than available processors. PUPPET has the ability to simulate multitasking, and so it allows such programs. This allows one to evaluate various load-balancing schemes.

## 1.3 Outline

The rest of the thesis is organized as follows. Chapter 2 surveys existing work done on parallel-computer simulation, as well as various processor and network simulators. In Chapter 3 we present the methods of PUPPET in a theoretical manner. We describe the implementation of PUPPET in Chapter 4. Chapter 5 gives results validating PUPPET and shows its usefulness in combination with performance-visualization tools. Finally, we conclude in Chapter 6 and overview possibilities for future work.

# Chapter 2

# Literature Survey

In this chapter we look at related work that has been described in the literature. Section 2.1 describes competing parallel-computer simulators and compares them to PUPPET. In Section 2.2, we examine simulators for components of parallel computers that have the potential of being used in conjunction with PUPPET. We discuss processor and network simulators in Sections 2.2.1 and 2.2.2, respectively.

## 2.1 Parallel-Computer Simulators

Many vendors have examined the use of simulation to evaluate the performance of parallel computers prior to building them. Researchers have also used simulation to evaluate new techniques for solving problems, in particular, cache coherency. Much of the development in parallel-computer simulation has targeted shared-memory systems, which are not of interest to us. Such simulators include Chief [12], MINT [77], SPASM [68], PSIMUL [70], Talisman [3], Tango [24], and

Threads [53].

Unfortunately, far less attention has been given to simulation of distributed-memory (message-passing) systems. The known projects that have addressed this problem are the topic of this section. They have a variety of common properties, and hence we cover common features instead of going into detail for each simulator. Similarly, we compare the individual features (instead of entire systems) to PUPPET.

### 2.1.1 Overall Organization

Most modern-day simulators involve *discrete-event simulation* (that is, are *event-driven*) [32]. This means that (a finite number of) events control the simulation; for example, their occurrence causes the advance through time. The simulator essentially "waits" for an event to occur, does some processing, and then continues. The times at which events occur are the only points when anything is done; these computations usually depend on the nature of the event. Most events are not known ahead of time: processing an event usually involves creating new events. The main advantage of discrete-event simulations is that events can be scheduled at any time.

An alternative to the event-driven approach is *discrete-time* (*time-driven*) [32] simulation. In this method, there is a global clock that all resources (e.g., processors) are consistent with. During each iteration, some operations are done, and the clock is incremented by a constant amount. One way to use this is to mark events with a *time-left* attribute which says how many clock ticks are left until the event occurs; when the clock is incremented, this value is decreased, and when it reaches zero the event is executed. This makes the simulation inflexible in terms of granularity of timings. In order to achieve complete generality, the clock unit would have to be extremely small, for example, a microsecond; this however would cause a large overhead if no events occurred during entire seconds.

If time cannot easily be discretized into multiples of a single value, then time-driven simulations are ineffective. For this reason, all simulators that we know of are based on the discrete-event technique.

### 2.1.2 Simulation-Program Coupling

Some simulations are designed to measure the performance of a parallel computer, and assume an artificial workload such as "delays between message-sends follow a negative-exponential distribution" (for example, a similar assumption is made in Calahan and Bailey [14]). On the other hand, we are interested in the case where the workload is generated by the user's parallel program. There are two ways that the program can describe the workload to the simulator, which involve different levels of coupling between the simulator and the program.

The first option, which is used in PUPPET, is to have a *loose coupling* (sometimes called a *trace-driven* simulation [31]). The program executes on an arbitrary *host*, such as a pseudo-parallel system, and records a log of its communication and (amount of) computation. Then as a post-processing phase, the execution is simulated with a particular parallel computer in mind. Note that none of the simulation parameters come into play when the log is generated. Therefore, various options can be tweaked and the program can be re-simulated, without needing to re-run it.

Assuming that the simulation is fast (as it is with PUPPET), a complex program can be evaluated for many platforms in little more time than required for a single platform.

An alternative is to make a *tight coupling* and in fact combine the simulator and the program. The program is instrumented either by the user or an automated tool so that relevant events are handled by the simulation portion. Various delays are made so that the program thinks that its environment is the simulated system. This has the obvious disadvantage that the simulation is not a post-processing phase and model changes require a complete re-execution. It does, however, have two main advantages that are worth noting.

First, no log files are created. For extremely long-running programs, loose coupling causes particularly large logs to be created. In such cases, however, it is likely well worth using loose coupling if even two sets of model parameters are to be tested.

Second, processes are allowed to be non-deterministic in the way that they communicate with others. For example, suppose that a process attempts to receive a message, but it may timeout. With a loose coupling, the process might timeout in the pseudo-parallel environment although it would not have in the parallel computer being simulated. On the other hand, a tightly coupled simulation would correctly detect whether a timeout occurred.

Fortunately, there are a wide variety of applications (e.g., most scientific codes) that are *network-deterministic*[1], that is, the timing of the various communication

[1]We add the "network" qualifier to avoid confusion with the non-determinism involved with random numbers.

events affects little in the program other than speed. In these cases, loose coupling applies well.

In the future, it would be worth considering the extension of PUPPET to have the option of being tightly coupled to the program. This would broaden the set of potential uses of PUPPET.

### 2.1.3 Method of Timing

The program's computation is timed by either a log-generator (for loose coupling) or by the simulator itself (for tight coupling). In either case, the way in which we measure the computation cost of a particular block of code is important. There are three basic techniques that are commonly used in simulators, each with its own advantages and disadvantages.

*Instruction-level* simulations are the most accurate possible. They use processor simulators (see Section 2.2.1) to simulate each instruction of the program, thereby determining the time to execute a particular group of instructions. This allows complicated factors such as caches and the speedup of vector operations to be accurately measured. Unfortunately, instruction-level simulations cause programs to speed-down by around 300 times [71]. It also takes significant effort to build a simulator for a particular processor, and building a portable instruction-level simulation is almost impossible. Since the overhead of instruction-level simulations is so high, it does not make sense in combination with loose program-simulation coupling.

Another possibility is to time computation using CPU time, such as the facility provided by UNIX (sometimes called an *execution-driven* simulation[2]). The advantage of this method is that it is extremely portable and easy to implement. However, most UNIX systems limit their accuracy to multiples of ten milliseconds. This can be insufficient for very fine-grain programs. The method also requires that the program run on a processor identical to one of the simulated parallel computer.

Much of the recent simulation development has been towards *program augmentation*, which is a compromise between the previous two methods. Augmentation was first used by Threads [53], later refined in RPPT [19], and perfected in Tango [24] and Proteus [8]. The idea is to count (at compile-time) the number of processor clock cycles required for each *basic block* (a block of code without any control flow or communication instructions) by examining the involved instructions. The code is then instrumented so that the simulator is notified of any computation done in terms of clock cycles. This fails to take into account caches and vector operations (as the instruction-level approach did), although it is more accurate than CPU time. Augmentation is faster and more portable than instruction-level simulations, although it still takes a fair amount of effort to support a wide variety of processor architectures.

Other advantages of augmentation include non-intrusive debugging and profiling. The simulator will not notice any differences if the program is a little less efficient because of adding profiling or other data-collection code used for debugging, provided that the basic blocks of the added code are ignored by the augmenter. This is in contrast to a parallel computer (or any live system) where the *probe ef-*

[2]Unfortunately, the augmentation method is also sometimes called execution-driven [20].

*fect* [35] may cause a phenomenom to disappear when one attempts to measure it. In addition, augmentation allows unfortunate errors such as stack overflow to be detected and reported in an efficient way [7].

The above three advantages are also possible with instruction-level simulations. In fact, the most common use of processor simulators is to exactly evaluate performance of the processor, avoiding the probe effect. Stack overflow is also easy to detect and report, although it cannot be done as efficiently as augmentation; the stack pointer must be checked after each instruction instead of each basic block[3].

Currently, PUPPET uses the CPU-time approach. Given time to do so, augmentation could also be implemented. Augmentation essentially provides the same information as CPU time, but with higher accuracy.

### 2.1.4 Network-Simulation Accuracy

In addition to processor-timing accuracy, we can also examine the amount of accuracy obtained in simulating the network. The most common method is to employ a network simulator (see Section 2.2.2). This allows the measurement of all network latencies, including routing, traffic, and congestion (i.e., hot spots). Unfortunately, network simulations often take a significant amount of time. Thus, the major advantage is accuracy and the major disadvantage is speed.

The obvious simplification is to use an analytical model to approximate the latencies and improve performance. Agarwal [1] proposed a model for a wormhole-

[3]It is possible to improve on this. For example, provided that the stack-pointer is at least 100 units from overflowing, the processor simulation could continue for 100 units before checking (or some fraction thereof, depending on what kinds of instructions are available).

routed $k$-ary $n$-cube interconnection network (an $n$-dimensional $k \times k$ torus) embedded into two-dimensional space. The model attempts to take into account congestion based on the frequency of past requests, providing speedups of between two and four over a detailed network simulation [9]. Unfortunately, such models do not yet exist for all networks, and it may be difficult or even impossible to create models with reasonable accuracy for some networks.

Another approach, proposed in Sections 3.2 and 3.3, is to ignore congestion caused by multiple messages. In other words, the arrival time of a message is calculated as if the network were otherwise empty. This method yields immense performance benefits and (as we shall see in Section 5.2) often maintains considerable accuracy. Currently, it is the method used by PUPPET, but it is certainly possible to extend PUPPET to allow the other methods to be used.

Such an ability to vary accuracy would be extremely useful. It would allow one to get a rough idea of performance during debugging and testing, and also get high accuracy with the flick of a switch. If a loosely coupled simulation is used, this can even be done without re-running the program.

### 2.1.5 Language

An important human factor of parallel-computer simulation is its ease-of-use. A significant part of this is the programming paradigms (i.e., *languages*) that the simulator supports. The best scenario is that the program is written in a supported language; for example, an MPI program to be used with PUPPET. Usually, little user effort is required to simulate fully supported programs. Popular message-passing environments include MPI and PVM (Parallel Virtual Machine) [37]. In addition, programs written in parallel languages (which eventually translate into message-passing programs) may be used; a small sample of them consists of HPC (High Performance C) [30], HPF (High Performance Fortran) [51], and pC++ [4].

Unfortunately, simulators do not always directly support a popular parallel-programming language. This implies that the user must first spend a certain amount of time learning the language. If the parallel program is already written, it must be converted to that language. These steps can be very tedious and should be avoided whenever possible.

### 2.1.6 Parallelism and Concurrency

Our final consideration is parallelism present within the simulator. In simulating a parallel program, some authors claim that there is inherent parallelism available, which can be exploited to yield good speedup [26, 71]. The idea is that the simulator needs to use communication only when the program's processes communicate. Essentially, the parallelism from the program is carried over to the simulator.

For tightly coupled simulation systems, this makes sense: part of the overall system is the parallel program. Parallelization is effective since typically the overhead is in the computation-measurement process described in Section 2.1.3.

It is questionable how useful a parallel simulator is, since parallel resources are rare. However, networks of workstations (NOWs) and workstation clusters are becoming increasingly popular; even having two processors to simulate many would be beneficial. Unfortunately, because of several complications (e.g., a mix of both operating-system multitasking and hand-written multithreading), few tightly coupled simulators allow parallelism, and those that do usually sacrifice some efficiency in the process.

On the other hand, loosely coupled systems can automatically exploit parallelism. This is because the simulation is broken up into two phases: the parallel program execution (which is parallel) and the post-processing simulation (which is serial). The execution includes the major source of overhead mentioned in Section 2.1.3, namely computation measurement, and there is no reason that the execution cannot use a parallel system. Hence, for these systems (such as PUPPET), parallelism can already be easily achieved.

An issue related to parallelism, briefly mentioned above, is how concurrency is provided. The simplest option is to allow the operating system to see the processes, and allow it to multitask them. This often yields a very high overhead, although it allows parallelism to be accomplished (for example, UNIX processes can execute on multiple processors). An alternative is to provide custom multithreading code, greatly improving task-switching latency and message-passing efficiency. However, this method requires that the entire program be run in pseudo-parallel and so performance enhancement via parallelism is not possible.

PUPPET does not necessarily take either option; rather, it depends on the underlying MPI implementation. To date, all UNIX-based MPI implementations use UNIX processes; however, it is quite possible that a multithreaded implementation could be developed. Such an effort has been made by Villanueva [75] for PVM. The results are very impressive: 50,000 processes can be used efficiently on a personal computer.

### 2.1.7 Survey

We shall now briefly describe various distributed-memory parallel-computer simulators, relative to the above considerations. Their characteristics are summarized in Table 2.1.

The *Rice Parallel Processing Testbed* (RPPT) [19] was developed at Rice University. It is based on top of the parallel language *Concurrent C* [52], also from Rice, which provides multithreading support. The Parallel Tracer/Debugger (TRAPP) [20] allows graphical debugging of the program.

*Proteus* [8] is the most advanced parallel-architecture simulator, developed by Brewer and Dellarocas at M.I.T. for their Master's theses [7]. The authors claim it to be "fast, accurate, and flexible" [9]. It achieves its speed partly from augmentation and its built-in threads, and partly from its module system (which also provides flexibility). The idea is that there are several implementations for each task (such as network simulation), each with its own level of accuracy and efficiency. Proteus also provides some useful graphical utilities for generating plots of performance data.

The *EPPP simulator* [63] is an extension of Proteus for *High Performance C* (HPC) [30] programs. Even though it is augmentation-based, the simulator attempts to evaluate pipelining and vector operations. The EPPP (Environment for Portable Parallel Programming) project [47] provides other tools for HPC, including a performance visualizer and debugger. It is based at the Centre de Recherche

| Simulator | Coupling | Timing method | Network sim. | Language | Parallel | Threads |
|---|---|---|---|---|---|---|
| RPPT [19] | Tight | Augmentation | Accurate | Concurrent C [52] | No | Yes |
| Proteus [8] | Tight | Augmentation | Accurate; Agarwal | Custom | No | Yes |
| EPPP [63] | Tight | Augmentation | Accurate; Agarwal | HPC [30] | No | Yes |
| LAPSE [26] | Tight | Augmentation | Accurate | NX | Yes | No |
| PAPS [78] | Loose | None | Accurate | Task graph | No | — |
| PerPreT [5] | Loose | Analytic | Accurate | LOOP [6] | No | — |
| ExtraP [57] | Loose | CPU time | Accurate | pC++ [4] | No | Yes |
| APNM [67] | Either | Augmentation | No traffic | PVM | No | Varies |
| PUPPET | Loose | CPU time | No traffic | MPI & NX | No | Varies |

Table 2.1: *A summary of various distributed-memory parallel machine simulators, including PUPPET. The "overall organization" is consistently discrete-event, and hence omitted. For loosely coupled simulations, "non-parallel" means that the second phase (simulation) is serial.*

Informatique de Montréal (CRIM), McGill University, and Concordia University.

*LAPSE* (Large Application Parallel Simulation Environment) [26], developed by Dickens, Heidelberger, and Nicol, focuses on parallel simulation. In general, they find reasonably good speedup, although it greatly depends on the amount of parallelism in the simulated program. LAPSE is currently based on the Intel Paragon multicomputer, which uses the NX message-passing library.

*PAPS* [78, 79] is a simulator developed by Wabnig and Haring at the University of Vienna. It uses Petri nets, a model of parallel processes, to model the network. To do this, it enforces that the "programs" are specified as task graphs, an abstract view of the components involved in the computation and their ordering. This essentially corresponds to a log of a program execution, thereby making it loosely coupled. Task graphs make PAPS restrictive in the types of communications allowed. The generation of the task graphs is not a part of PAPS.

*PerPreT* [5], created by Brehm, is also based on a task-graph model of programs, but does not limit the communication forms as much as PAPS. Programs may be coded in a language called LOOP [6], also written by Brehm. The computation portions are then analyzed statically, and hence the program must have a static pattern. Conditionals do not exist, although loops are allowed. Some parallel algorithms, such as conjugant-gradient, are programmable in this way, provided the number of iterations is constant.

*ExtraP* [57] is a *pC++* [4] tool for extrapolating pseudo-parallel performance to parallel performance from the University of Oregon. It is a part of the $\tau$ (TAU, Tuning and Analysis Utilities) tool-set [11], which includes several performance-visualization and analysis programs for pC++. A graphical interface to ExtraP is provided by *speedy* (Speedup and Parallel Execution Extrapolation DisplaY).

After the implementation of PUPPET was completed, we became aware of *APNM* (Abstract Parallel Numerical Machine) [67] which was developed at TUM (Technische Universität München) simultaneously with and independently from PUPPET. It uses a network model along the same lines as ours, but they ignore network topology, hence also ignoring switching methods. Furthermore, they do not support collective operations. Both PVM and NX (the language of the Intel NX multicomputer) programs are supported. A feature unique to APNM is that either loose or tight coupling can be used.

## 2.2 Simulating Components

Significant work has been done on simulating the two main components (processors and networks) of parallel computers independent of the rest of the system. The following two sections overview proposed systems.

### 2.2.1 Processor Simulators

We are interested in processor simulators because they offer insight into the computation portion of a parallel program. Traditionally, they have been used for simulating workstations, which allows exact benchmark measurement. Six major systems have been proposed, allowing simulation of certain processors designed by HP, SUN, and DEC.

DEC's *SPIM* (MIPS spelled backwards) [61, p. A-36] supports MIPS

R2000/R3000 processors. It includes an X-windows debugger, since powerful debugging is the main focus of SPIM. Programs typically run 100 times slower than using real MIPS processors.

*MINT* (MIPS Interpreter) [77] simulates MIPS R3000 and some MIPS R4000 instructions, although it has a flexible design so that it can be extended for other architectures. Several host platforms are supported, including SGI, DEC, and SUN SPARC workstations. It uses a mix between executing instructions on the host system and simulating the instructions to achieve efficiency. MINT also includes support for shared-memory multiprocessors that use a MIPS instruction-set.

*Paint* (PA Interpreter) [73] is an instruction-set simulator for the PA-RISC architecture [18]. It is based on MINT, illustrating MINT's flexibility; this has the effect that shared-memory multiprocessors are also supported in Paint. Although it is under a different name, Paint is essentially an extension of MINT.

*SimICS* [65] is a system-level interpreter for the SPARC V8 [72] instruction-set and architecture. It typically achieves a slowdown factor of 15 when running on other Sun architectures. SimICS's goal is to run an operating system; currently, it only supports a UNIX-compatible mode.

*Shade* [16, 17] simulates SPARC V8 and V9 architectures, as well as the MIPS I instruction set. The host machine is forced to be a SPARC, as with SimICS — this is rather unexpected for the MIPS simulator. Shade introduces dynamic compilation and caching techniques for fast cross-architecture simulation. Slowdowns range from 6.1 to 31.2, depending on the level of output.

*Mable* [23] is a general framework for instruction-level machine simulation. Typical slowdown factors range from 20 to 200, depending on the detail of simulation. It has been used to develop three processor simulators: Multiprocessor (MP) Mable, designed for shared-memory multiprocessors using the MIPS-III instruction set; Tsim, a simulation of the TORCH processor [69] which supports speculative execution; and PPsim, a simulator for the FLASH shared-memory multiprocessor.

In general, processor simulators provide high accuracy but have significant overhead. They are particularly useful when effects such as vectorization and caching play crucial roles in the performance of parallel programs. PUPPET currently does not support any processor simulators, although it could be extended to do so.

### 2.2.2 Network Simulators

Independent interest has also been spawned in the area of networks. Network simulators are typically designed to evaluate the performance of a network subject to some randomly generated message patterns. This is particularly useful for design of new networks; simulators can be used to evaluate metrics such as average latency under a particular load. Similarly, simulations can be used to evaluate new routing algorithms. Network simulators can also be used for overall parallel-computer simulation (our interest), since they evaluate delays involved in sending messages.

*NETSIM* [49] is designed for all three uses. It forms a part of RPPT (see Section 2.1.7), as well as serving as an independent network simulator. NETSIM supports direct and indirect (multistage) interconnection networks. A routing algorithm can be specified by the user, using one of packet switching, virtual cut-through, or wormhole routing techniques.

*NetSim* [42], although it has a name clash with NETSIM, is entirely different. It is designed for simulating circuit-switched networks (both direct and indirect), which is the one switching mechanism that NETSIM does not support. NetSim has also been used to simulate a reconfigurable multicomputer network called the Interconnection Cached Network (ICN). The load of a simulation is measured by a single parameter, $m$, the probability of sending a message in each time-step.

*PP-MESS-SIM* [64] is the most recent network simulator. A key feature is that it is object-oriented (via C++), making it very flexible. It supports the same switching techniques as NETSIM, namely packet switching, virtual cut-through, and wormhole routing, and allows the $k$-ary $n$-cube direct interconnection network [21]. Network workload is only slightly more complex than NetSim; for example, a negative-exponential distribution of send operations can be used.

*PARSE* (Parallel ARchitecture Simulator Environment) [60] is meant to help design networking hardware in distributed-memory supercomputers, although it allows simulation of a general (SPMD) program. It uses a very detailed model, going down to the data-link layer, including the flow-control protocol, and also examines factors such as copying buffers in routing. As Olk [60] says, it "properly model[s] all performance aspects."

*XPOSÉ* [81] is designed for local-area networks. It would be useful for simulating a network of workstations when many networks are involved. These networks must be primarily buses, connected by various links. It was developed to facilitate the development of hardware as well as software.

It seems that much of the research in network simulation has been repetitive.

Although the above five systems each have distinct features, there is a large amount of intersection. In the future, we hope that PUPPET can bring together the available features, offering (as an option) a detailed simulation of the network.

# Chapter 3

# Proposed Evaluation Technique

This chapter describes the models and theory that PUPPET is based on. Section 3.1 examines the portions of a parallel program running in a pseudo-parallel environment that should be timed, defining an execution log. In Section 3.2 we describe an overall model for the underlying network and the performance of the network links. Sections 3.3 and 3.4 detail this model, showing how to estimate the latency involved in point-to-point and collective communications, respectively. Finally, in Section 3.5 we see how the entire model can be simulated using discrete-event techniques, including details on supporting multitasking.

Before we begin, we make note of implicit assumptions in our reasoning. These essentially specify the target applications of PUPPET. Obviously, we are only interested in distributed-memory parallel programs, in particular, those that use a message-passing paradigm. Second, we assume that processes consist of a series of computation and communication instructions; we ignore any overhead such as that associated with I/O. Finally, it is assumed that the parallel applications are network-deterministic (see Section 2.1.2 for a definition).

## 3.1 Timing a Pseudo-Parallel Program

When dealing with programs running in a dedicated parallel environment, the basic metric used for further analysis is the total execution time. In other words, this is the maximum amount of time that a processor takes to complete, including both the communication and computation time. From this we normally calculate other metrics such as speedup. Parallel-execution time must be measured using the wall (real-time) clock, or else synchronization and communication factors will not be properly taken into account.

Such a simple evaluation method cannot be used when the program is run in a pseudo-parallel environment. First, the computation portion stays similar to the serial case because each pseudo-processor obtains one-$n$th of the CPU power, where $n$ is the number of pseudo-processors, assuming that a fair time-sharing algorithm is employed. Second, the communication portion is not accurate, since a network is not used to transfer messages. Instead, the operating system replaces communication primitives with shared-memory references. Finally, any context-switching overhead is included in the measure. Similarly, with parallel systems shared by multiple users, the processors and network appear weaker and hence the timing is inaccurate.

This problem is not unique. When time-sharing uniprocessors became common and multiple programs were run simultaneously, it was difficult to determine the amount of time that any one program actually took, in other words, the amount of time that the program was computing and not waiting for its time-slice. For this reason, the concept of CPU time was introduced, measuring the amount of time that a program occupies the CPU, that is, computes.

We wish to create the analog of CPU time for pseudo-parallel programs. A first attempt would be to measure the CPU time spent by each pseudo-processor. Unfortunately, this has the effect of incorrectly measuring communication time. We would hope that this measure included only computation, so that in a second phase we could incorporate communication time. However, communication is achieved using shared-memory references (possibly through a TCP/IP stack), which can take a significant amount of CPU time.

Instead, we must be careful not to incorporate the time taken by the underlying message-passing layer. To do this, we can instrument the program so that we stop (start) timing immediately before (after) each call to a communication routine. Hence, we have timed the *computation blocks*, each corresponding to a communication call (Figure 3.1). By convention, we associate the computation block immediately preceding a communication with that communication.

In the remainder of this chapter we will see how the computation blocks, along with details of the communication calls (together called an *execution log*), can be used to simulate the parallel program. In this way, we can estimate parallel execution of the program as if it were run on a specified parallel computer. First we will look at how the parallel computer's network can be specified (that is, modeled), and later at how it can be simulated.

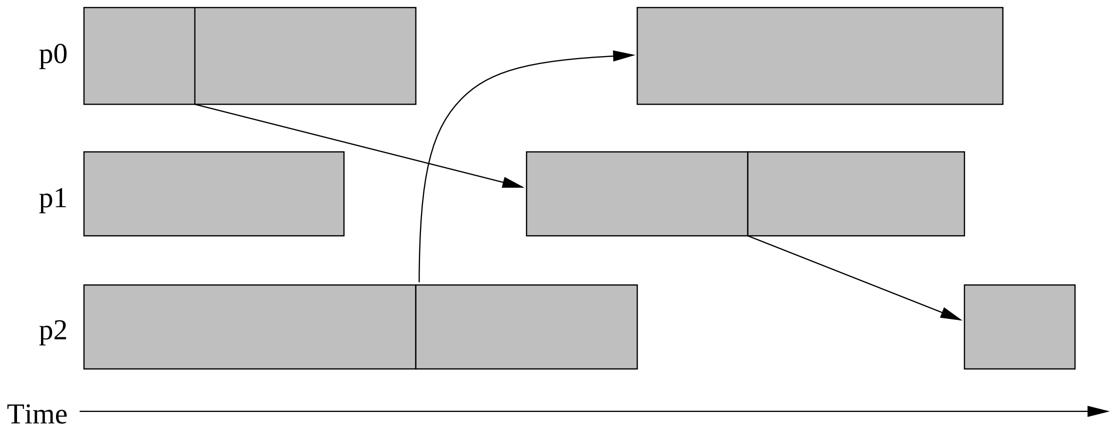


Figure 3.1: *A sample parallel-program execution. The shaded rectangles represent computation blocks. Blank spaces represent blocking (waiting for a receive to complete). Arrows denote messages.*

## 3.2 Network Simulation

A fundamental part of a parallel computer is the network, which is used to transmit data between processors. Network simulation is a large area of research (see Section 2.2.2). The standard approach is to monitor the (simulated) travel of each message, including what processors and other hardware (such as switches) it encounters. Each encounter creates a certain amount of delay, and after a certain amount of time the message arrives at its destination.

Such a simulation also takes note of any conflicts when multiple messages require the same piece of hardware. In other words, the message delay caused by *network congestion* is taken into account. Although this delay affects parallel execution, much congestion is usually required before the program is slowed down significantly. In addition, reaching this (maximum) amount of accuracy greatly increases simulation time.

We take the view that simulation should be a post-processing phase. Hence, one could vary model parameters (that is, the specification of the simulated parallel computer) several times but only run the program once. This allows one to ask several questions such as "what if the network was twice as fast?" or "what if I added a bus for broadcasting data?" If we achieve efficient simulation, such questions can be answered almost instantaneously.

We thus set out to create an efficient simulation of the network that does not take into account network traffic or congestion. In many cases (as we shall see in Sections 5.2 and 5.3), this does not significantly affect accuracy and leads to an immense performance benefit.

### 3.2.1 Overall Model

We assume that the network is *direct* (or *static*) so that each *network link* allows exactly two processors to communicate. Hence, we can represent the network as a connected directed graph $G = (V, E)$, called the *network graph*, where the vertices (edges) correspond to processors (links). Typical topologies (i.e., graphs) include meshes, hypercubes, tori, and rings [62]. Since we model the network as a *directed* graph, the links need not be bi-directional.

Next we shall examine how we can model the performance of individual links. For simplicity we assume that each link is identical with respect to speed. (This is almost always the case except in heterogeneous systems, which are not considered in this thesis because of their many complications.)

### 3.2.2 Network-Link Model

The goal of network links is to achieve transmission times proportional to the message size. Unfortunately, current technology limits us to achieving this only when messages are sufficiently large. When small messages are sent, the dominating factor is the *initialization time* that it takes to prepare the link for transmission.

Another complication of link performance is *packets*. In many networks, messages are split up into fixed-size packets each of which is transmitted separately. For example, Ethernet-based networks employ a packet size of 1500 bytes [58, p. 101]. Since each packet is transmitted separately, an initialization cost must be paid for each packet.

Therefore, we adopt the following model to estimate typical network-link performance. The parameters are $i$ (the initialization time per packet), $p$ (the size of a packet), and $t_1$ (the time to send one unit of information across the link). Given the size $k$ of the message, the total time to transmit the message across a single link is

$$t(k) = i \cdot \left\lceil \frac{k}{p} \right\rceil + k \cdot t_1.$$

In the case where packets are not employed, i.e., $p = \infty$, we must modify the formula to

$$t(k) = i + k \cdot t_1$$

so that initialization time is paid.

When extremely large messages (large than a megabyte) are involved, performance drops [66]. Since little is known about the exact behavior in this case, we suggest a simple generalization of the above model: let $t$ be a piecewise-linear function. This incorporates the idea of packets, as well as allowing non-monotonicity. Since large messages are somewhat uncommon, we have not yet implemented this extension in PUPPET.

## 3.3 Point-to-Point Communication

It is rare that the graph defining the network topology is completely connected, because of inherent cost restrictions. In this section we examine how to simulate the travel of a message through several links. For the moment, we assume that only one source node and one destination node are involved; in Section 3.4 this restriction will be removed.

### 3.3.1 Routing

The problem of routing a message through a network to avoid congestion, deadlock, livelock, and starvation has long been studied [25]. Many routing algorithms have been proposed for various networks [33, 36, 59]. Each can be defined as a possibly non-deterministic function yielding a path between two given nodes. Fortunately, in our less detailed simulation we do not need a routing algorithm. Instead, we assume that optimal routes are always taken; which route is chosen does not matter, since we are not considering congestion. In our case, the only identifying feature of a route is its *length*, that is, the number of links that must be traversed to send the message. The length of a shortest route between two nodes is called the *distance* and will be denoted by $d$.

Determining for each pair of processors $(\mathsf{src}, \mathsf{dest})$ the value $d$ is equivalent to the *all-pairs shortest path* graph problem. It can be easily solved in $O(n^3)$ time using the classical algorithms of Dijkstra [27] and Floyd [34], where $n$ is the number of processors. Using more complicated techniques, one can achieve $O\left(n^3 (\log\log n/\log n)^{1/2}\right)$ time [74] or $O(n^2 \log n)$ expected time [56].

### 3.3.2 Switching

Routing consists of more than determining a path from the source to the destination processor. The *switching technique* determines the way that messages travel along a chosen path. There are four common methods: packet switching, virtual cut-through, circuit switching, and wormhole routing, illustrated in Figure 3.2.

*Packet switching* (or *store-and-forward routing*) [33] is the most intuitive switching method. A message being transmitted simply travels over all the links on the path, reaching intermediate processors as it goes. The processors are interrupted and determine which way the message should go next, and then transmit it to the next processor, until the destination is reached. At any point in time, only one link in the path is reserved for a single message. Packet switching offers good solutions to deadlock and other problems. It has the severe disadvantage that latency (the time to reach the destination) is proportional to $d$.

To improve packet switching, *virtual cut-through* [50] was proposed. In this method, special switching hardware is employed so that a message will only inter-

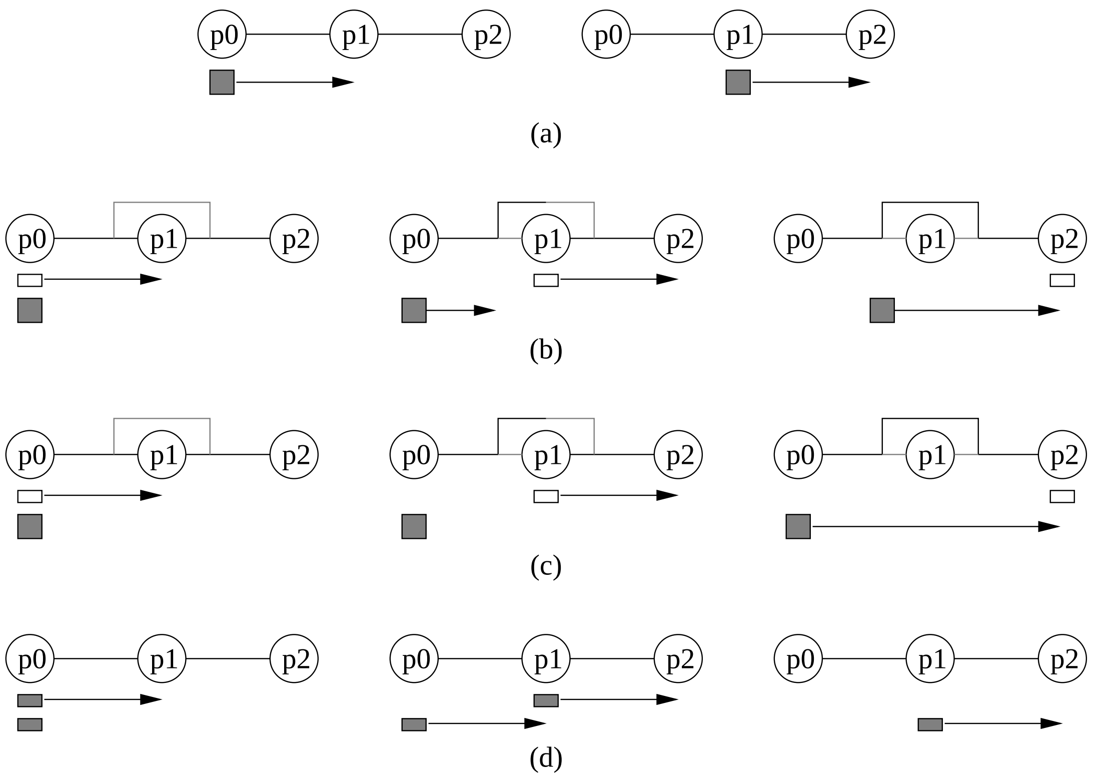


Figure 3.2: *Illustration of the various switching techniques on a 3-processor linear array. (a) Packet switching. (b) Virtual cut-through. (c) Circuit switching. (d) Wormhole routing. Filled squares represent the entire message. Filled and empty rectangles represent flits and headers, respectively. Gray lines denote inactive links.*

rupt a processor when the next required link is busy (or the destination has been reached), sending a header of size $h$ to set up the hardware. The message trails behind the header. Virtual cut-through improves the performance so that for messages much longer than the header, $d$ does not affect latency. In particular, in the latency formula, the distance $d$ and $t(s)$ (the actual communication time) are not in the same term.

Two other methods have been proposed to achieve this. In *circuit switching* [41], a control packet of size $c$ is sent through the network to reserve all of the links on the path. As soon as they have all been reserved, special hardware allows the links to combine and become a single "circuit" that the message can be sent through at a speed similar to that of a single link. Circuit switching is similar to virtual cut-through except that the message waits for the header to finish reserving instead of trailing behind it. If we ignore congestion, circuit switching ends up using the same amount of time as virtual cut-through by letting $c = h$ [59].

The final method, *wormhole routing* [22], is often accepted as the best. Each message is split into a sequence of fixed-size *flits* (flow control units); flits are typically only a byte or two long. The flits are then sent in a pipeline manner along the path. An important point is that each flit must follow the same path; this way, less initialization cost is likely needed for all flits but the first. As with typical pipelining, if the number of inputs (flits) is sufficiently larger than the number of operations (links), then the number of operations (links) does not affect the total time. Thus with extremely small flit sizes, a great performance benefit is obtained.

We summarize the time taken by the switching techniques in Table 3.1.

| Switching technique | Latency when there is no congestion |
|---|---|
| Packet Switching | $d \cdot t(s)$ |
| Virtual Cut-Through | $d \cdot t_0(h) + t(s)$ |
| Circuit Switching | $d \cdot t_0(c) + t(s)$ |
| Wormhole Routing | $(d - 1 + \lceil s/f \rceil) \cdot t(f)$ |

Table 3.1: *The latency of the various switching techniques. d is the distance between the source and destination. c, h, and f are the sizes of the control packet, header, and flit, respectively.* $t(k)$ *is the time to send a message of size k across a link (see Section 3.2.2);* $t_0$ *is the same but may have different parameters than t. s is the size of the message.*

Systems that apply packet switching include the Cosmic Cube, iPSC-1, Ncube-1, Ametek 14, and FPS T-series [59]. The University of Michigan's HARTS (Hexagonal Architecture for Real-Time Systems) multicomputer [28] uses virtual cut-through as its switching method. Circuit switching is used by the Intel iPSC-2 and iPSC/860 multicomputers. Multicomputers that use wormhole routing include the Ametek 2010, Symult 2010, Ncube-2, Intel/DARPA's Touchstone Delta, Intel Paragon, MIT's J-machine, Intel/CMU's iWarp, and the Transputer IMS T9000 family.

## 3.4 Collective Communication

*Collective communication*, in contrast to point-to-point communication, involves several, potentially all, processors in the network. Whereas point-to-point communication has essentially one basic form (send-receive), there are several operations that are collective. We shall consider the following:

- *Barrier synchronization*: certain processes cannot complete until they have all reached the barrier.
- *Multicasting*: send the same message from one process to several others.
- *Scatter*: send different messages from one process to several others.
- *Gather*: send messages from several processes to one process.
- *All-to-all*: several processes send messages to every other process in the group.
- *Reduce*: operations such as global maximum. One variant leaves the result at a single process, whereas the other gives the result to all involved processes.

As with the point-to-point case, we will examine models that ignore network congestion. If more detail is required, a network simulator (such as those described in Section 2.2.2) can be used.

### 3.4.1 Multicasting

There are three general techniques employed in parallel systems to send a message $m$ from process 0 to processes 1 through $n$:

1. *Obvious*: process 0 issues $n$ send operations, and the remaining processes issue receives. This technique is really only useful if the send operations are buffered, so that process $i$ need not receive before the message even starts traveling to process $i+1$.

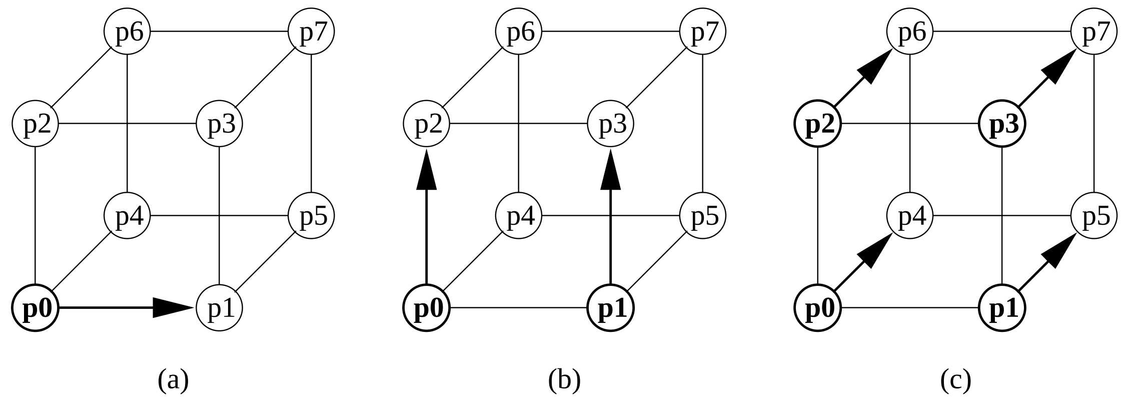


Figure 3.3: *An example of the smart multicasting approach: the hypercube broadcast algorithm [48]. During a phase, each processor that has the message (those in bold) sends in a particular direction (shown with arrows), which varies over time. (a–c) show time steps 1–3, respectively, for a three-dimensional hypercube.*

2. *Smart*: in some prescribed manner depending on the network topology, the message propagates from process 0 to involved processes on the same or adjacent processors, eventually reaching all of the destinations (Figure 3.3).

3. *Extra hardware*: a bus, hierarchical bus, or other additional hardware is present to allow faster multicasting.

Analogous to the point-to-point case, the problem is much simpler when congestion is ignored. In fact, the obvious and smart approaches are nearly equivalent: the inefficiency of the obvious approach, that the multiple messages cause high congestion and may collide, is nullified.

The extra-hardware approach can be considered to be the obvious approach, except that a network other than the one for point-to-point communication is used. In the case of a bus, this network is completely connected (that is, processor distances are one). We can view a multi-level bus as a tree network (Figure 3.4) with

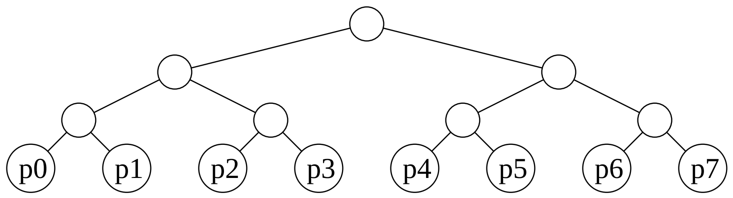


Figure 3.4: *A complete binary tree multicast-network. This can be used as a multi-level bus, or (if the link speeds are doubled) as a smart approach using a recursive structure. Smaller circles denote dummy processors.*

internal nodes that are *dummy processors*, that is, processors with no processes running on them.

Therefore, the general model that we use takes the obvious approach, where the sender issues many (buffered) sends, and the receivers issue typical receives, but a separate *multicast network* is used for this kind of communication. This may have a different topology, link speed, and switching technique compared to the point-to-point network. Potentially, the two networks are the same, in which case the model is simply the obvious approach. Note that the generated messages are not confused with usual point-to-point messages since they come from a different network.

The smart approach may in fact employ a recursive (tree) structure of communication (such as that in the hypercube algorithm in Figure 3.3), which is entirely different from the obvious approach. In this case, the general model still applies if we use a complete binary tree as the multicast network (Figure 3.4), where only the leaves are not dummy processors (giving the desired recursive structure), and the links have twice of the actual performance (so that we do not count both the time to go up the tree and the time to go down).

### 3.4.2 Other Collective Operations

Similar reasoning applies to the scatter and gather operations: the various messages are simply sent to the appropriate destinations. Multicasting is a special case of the scatter operation, although nowhere in our solution did we use the fact that the same message was being sent. All-to-all communication can be viewed as $n$ scatter (or gather) operations, executed simultaneously. From this we can define a barrier synchronization to have the performance of an all-to-all communication of $k$ units, where $k$ is likely 0 or 1 (it simply represents a header) [54]. Finally, a reduce operation can be viewed as a gather, possibly followed by a scatter (if the result of the reduction is to be known on several processes). Improvements on this simply reduce the amount of congestion.

It is possible that a separate network is used for each type of collective operation, but this is rare because of the cost. In the model, however, this is useful. For example, suppose that an all-to-all communication takes $n$ times longer than the equivalent scatter operation (on a particular parallel computer), since it uses a bus. This could be modeled by specifying that the performance of the all-to-all network-links is $n$ times slower than those of the scatter network. In addition, the multiple networks allow easy distinguishing between various types of messages.

Now that we have a model of the parallel computer's network, we can form a simulation of this model.

## 3.5 Discrete-Event Simulation

In this section we describe how we can use discrete-event-simulation techniques for the model developed in previous sections of this chapter. In Section 3.5.1 we overview the simulation architecture. Section 3.5.2 gives full detail of the algorithm. Finally, Section 3.5.3 discusses how multitasking can be dealt with.

First, we shall overview the general idea of a *discrete-event simulation* [32]; more detail can be found in Section 2.1.1. The simulation is driven by the occurrence of events, which are the only cause of moving forward in time. The events are processed in the order in which they occur, and are stored in a priority queue called the *global event list*. Processing an event may cause states to change, and may even cause new events to be created. For example, a send event (where a process issues a point-to-point send command) would likely create a message-arrival event for some time in the future; a blocking receive event, if no matching message is waiting, would cause the process to block (a *state change*). The key point of discrete-event simulation is that it is not *time driven*, that is, there is not a global clock incremented by a constant amount as the simulation continues.

### 3.5.1 Overall Architecture

As one can see from the overview of general discrete-event simulations, there is a lot of detail to be filled in. To begin, we give a brief description of the overall architecture (Figure 3.5).

Throughout we have been careful to separate the terms "process" and "processor." This is because we would like to simulate multitasking, that is, allow multiple

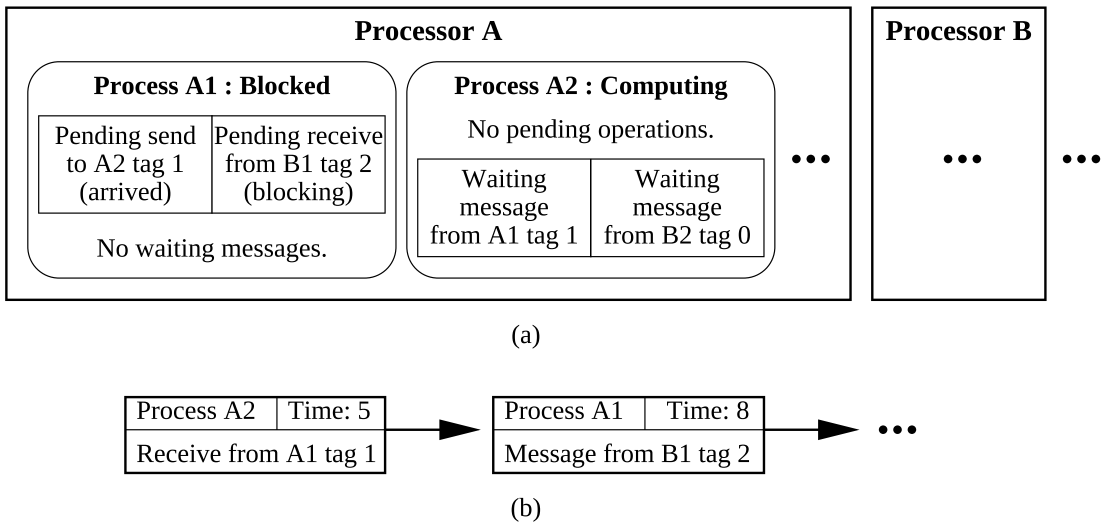


Figure 3.5: *Architecture of the discrete-event simulation, illustrated by a small example. (a) Process states. (b) Global event list.*

processes running on a single processor. Each process contains a single thread of execution, issuing various communication events. The timing of these events is not known prior to simulation; delays in blocking (which are only known during simulation) cause the times to vary.

Each process has a list of waiting messages. These are messages that have been sent by other processes (they may correspond to point-to-point, multicast, or other operations) but have not yet been matched with a receive operation. The message stores information such as the source process, the message tag (a value that must be matched on receipt), and the time at which it arrived. A message is added to the list even if the send operation which created the message is synchronous (in real life it would likely not take up resources at the receive-side), so that the receiver can easily identify any waiting messages, no matter how they were sent.

Each process also has a list of pending operations. They include any operations (such as point-to-point sends and receives) that have not yet completed or are otherwise of interest. This has several applications. For non-blocking operations, we can store a flag indicating whether or not the operation has already completed. Then when a wait command is issued by the process, we know if the process should be blocked. Waiting messages also store the sender's pending send operation that created the message. This way, the receiver can notify the sender of completion of the message-transfer by updating information about the operation.

A process has a state indicator. This specifies whether it is blocked (that is, waiting for one of its pending operations to complete) or whether it is computing (later to issue a communication request). Alternatively, a process may have exited (*dead*) or may not have been started yet (*prenatal*). If neither of these is the case, we call the process *alive*. Using this classification method, we can consider the process space to be fixed by starting with enough prenatal processes.

In addition, processes have an internal clock. This indicates how far in the simulation the process has progressed; since events are local to processes, they may not remain entirely synchronized. Whenever an event for a particular process is executed at time $t$, the process's clock is updated to $t$. The process's clock is useful to calculate the time that its next communication event will occur (the clock value plus the amount of computation prior to the event).

There are three main types of events that may be scheduled. The first (and simplest) is the issue of a communication operation. Such events are extracted from the execution log; they are scheduled throughout the simulation automatically, provided processes are not blocked (in which case they cannot issue any). There are three *operation-issue events*: point-to-point send and receive, and a wait operation (where a process pauses until a non-blocking operation is complete); collective operations are converted into the appropriate sends and receives. The second type is a *message-arrival event*, where a message moves into the waiting-messages pool (marked with its arrival time, that is, the time that the event occurred). Finally, an *operation-completion event* is used to tell a process that one of its pending operations has finished. For example, if the operation was blocking, this would cause the process to wake up.

We shall now turn to a more detailed description of the simulation.

### 3.5.2 Algorithm

The simulation consists of four main steps, aside from initialization (Figure 3.6). First, we take the next-occurring event from the global event list. Second, this event is processed, as we shall describe in the following few paragraphs. The last two steps are a kind of "cleaning up" for the next simulation step. We check for any waiting messages that match a pending receive operation on the same process, in which case we schedule an operation-completion event for the receive at the time that the message arrived or the time that the operation was issued, whichever is greater. Finally, we schedule operation-issue events for those processes that are not blocked and have no operation-issue events scheduled.

Processing an operation-issue event is reasonably straightforward (Figure 3.7). Suppose that the event occurred at time $t$ on process $p$. For send and receive events, the operation is copied into process $p$'s pending-operation list. For buffered

```
Procedure Simulation
    Initialize process states
    Schedule operation-issue events
    While the global event list is not empty,
        Remove the first event $e$ in the global event list
        Process event $e$
        Match waiting messages
        Schedule operation-issue events
```

Figure 3.6: *Pseudo-code for the discrete-event simulation. Details of event processing and scheduling of operation-issue events are given in Figures 3.7 and 3.8, respectively.*

sends, this operation is considered already done (on the send-side). A send event causes a message-arrival event to be scheduled on the destination process at time $t + N$, where $N$ is the induced network time under the model of Sections 3.2 and 3.3. Furthermore, if the send is synchronous and blocking, process $p$ is blocked. Similarly, if a receive operation is blocking, process $p$ is blocked.

Finally, we must consider how to process a wait event, where a process explicitly asks to wait for a previously issued non-blocking operation (now in the pending-event list) to complete. If the operation's done flag is set (e.g., it is a buffered send), then the operation is deleted and the process is not blocked. If this trivial case does not occur, the process is blocked and the operation is transformed into a blocking one.

Completion events allow processes to be unblocked if the operation previously caused the process to block. Furthermore, if the operation was a receive, it can notify the corresponding send that the message-transaction is complete.

For the moment, let us ignore multitasking. Then scheduling new operation-

```
Procedure Process_event $e$
    Advance $e$.process.clock to $e$.time
    Case $e$ is a send event:
        Add $e$ to pending-operation list for $e$.process
        If $e$ is asynchronous (buffered), then
            The pending operation's done flag is set to true
            The pending operation is set to non-blocking
        Schedule a message-arrival event for the destination process
        If $e$ is blocking, block $e$.process
    Case $e$ is a receive event:
        Add $e$ to pending-operation list for $e$.process
        If $e$ is blocking, block $e$.process
    Case $e$ is a wait event:
        If the pending operation being waited for is not done, then
            Block $e$.process
            Convert the pending operation into a blocking operation
    Case $e$ is a message-arrival event:
        Add $e$.message to waiting-messages list for $e$.process
    Case $e$ is a completion event:
        If the operation is blocking, unblock $e$.process
        Otherwise, set the operation's done flag
        If the operation is a receive operation, then
            Process a send-completion event for the source process
```

Figure 3.7: *Pseudo-code to process an event (non-multitasking version). Explicit garbage-collection details are not given for simplicity.*

```
Procedure Schedule_operation-issue_events
    For each process $p$,
        If $p$ is not blocking and has no operation-issue events scheduled, then
            Schedule the next operation-issue event using $p$.clock
```

Figure 3.8: *Pseudo-code to schedule operation-issue events (non-multitasking version).*

issue events is easy (Figure 3.8). We simply look for unblocked needy processes, and schedule operation-issue events for them. The time at which the event will occur is the value of the process's clock plus the amount of computation before that event (stored in the execution log). Note that this is not true when multiple processes execute concurrently on a single processor, since then a block of computation is not measured in real time but rather in CPU time. We will consider this in the next section.

### 3.5.3 Multitasking

Let us consider the case where potentially several processes share a single processor. We shall only consider the case where the processes have equal priorities and are scheduled in a round-robin fashion. We do not take into account any task-switching latency, thereby assuming that it is negligible. For purpose of discussion, assume that there is only one processor; this makes it easy to identify "the other processes on the same processor" (they are simply all other processes).

Multitasking complicates the simulation in two major ways. We need to modify the way in which completion events are processed (since they can cause processes to unblock), and the way in which we schedule operation-issue events. In fact, we do not want one operation-issue event scheduled per process, since the completion of one of these (possibly blocking the process) may change the occurrence times for the operation-issue events of other processes.

Instead, we only want to schedule one operation-issue event per processor. We store the next operation-issue events for the other processes (if any) with the pro-

```
Procedure Schedule_operation-issue_events
    For each process $p$,
        If $p$ is not blocking and has no operation-issue events in the
                processor's list, then
            Add the next operation-issue event to the processor's list
    For each processor $P$,
        If no processes on $P$ have operation-issue events scheduled, then
            Using $r$ to find the occurrence time, schedule the operation-issue
                    event in $P$'s list with smallest time
            When the operation-issue event occurs, advance processes
                    to that time
```

Figure 3.9: *Pseudo-code to schedule operation-issue events (multitasking version). $r$ is the number of running processes.*

cessor, in a list sorted by time. The one that we schedule is the first in the list. In addition, whenever we schedule a new operation-issue event, we must advance the clocks of the other processes to the scheduled time, and change the time at which their next operation-issue events will occur. In other words, whenever some computation is done to complete an operation-issue event, we must do the same amount of computation on the other processes, since they are all running concurrently. Thus, computation takes $r$ times longer, where $r$ is number of running processes. Figures 3.9 and 3.10 summarize.

Before $r$ changes, all computation that occurs before the change must be completed, so that the overhead of multiple processes can easily be measured. If $r$ varied over a block of computation, it would be difficult to calculate the total (real) time to execute the block. We handled the decrementation of $r$ by changing Procedure Schedule_operation-issue_events. $r$ is incremented whenever a process is

```
Procedure Advance_processes to $t'$
    For each other process on the current processor,
        $t \leftarrow p$.clock
        Advance $p$.clock to $t'$
        Decrease the computation (CPU) time prior to $p$'s next
                operation-issue event (if it exists) by $(t' - t)/r$
```

Figure 3.10: *Pseudo-code for advancing a process's computation to a specified time.*

```
Case $e$ is a completion event:
    If the operation is blocking, then
        If event is scheduled for processor, then
            Unschedule it and place in processor's list
        Advance processes to $e$.time
        Unblock $e$.process
    Otherwise, set the operation's done flag
    If the operation is a receive operation, then
        Schedule a send-completion event for the source process
```

Figure 3.11: *Modification of completion case for Procedure Process_event to allow multitasking. The only modification is the addition of the call to Advance processes.*

unblocked, which only happens during the completion of an event. In this case, we must advance the clocks of the other processes to the completion time, and change the time at which their next operation-issue events occur, using Procedure Advance_processes. Furthermore, the next event occurring on the processor may change because of a new running process, so we must unschedule such an event if it exists. The modification to Procedure Process_event is given in Figure 3.11.

Although all of this is rather complicated, multitasking is an important feature. It greatly increases the set of applications on which the simulator can be used, since many programs use concurrency in addition to parallelism. For these programs, various other factors, such as load-balancing schemes, can be varied and optimized, using the simulator.

# Chapter 4

# Implementation

An important part of this work is the implementation of the PUPPET system, described generally in the previous chapter. In this chapter we give details on the implementation. Section 4.1 provides a correspondence between MPI routines and the operations of Chapter 3. In Section 4.2, we discuss how MPI profiling libraries were used to automatically instrument programs. We describe the user interface to the simulator in Section 4.3. Section 4.4 looks at our use of ParaGraph, a parallel-program performance-visualization tool. Finally, we describe the network evaluator in Section 4.5.

## 4.1 MPI

The MPI 1.1 standard [55] defines an interface to 128 message-passing routines, categorized in Table 4.1. The only ones that are of interest to us (for simulation) are those that perform actual communication (dynamic process management is not

| Chapter | Category | Count |
|---:|---|---:|
| 3 | Point-to-point communication | 52 |
| 4 | Collective communication | 16 |
| 5 | Groups, contexts, and communicators | 30 |
| 6 | Process topologies | 16 |
| 7 | Environmental management | 13 |
| 8 | Profiling interface | 1 |

Table 4.1: *Overview of the MPI standard by chapter. "Count" specifies the number of routines defined in that category (chapter).*

provided by MPI). There are 32 and 16 of these in Chapters 3 and 4, respectively, of the MPI standard. Many of them correspond to the operations that we defined in Chapter 3. This section describes this correspondence for operations supported by PUPPET.

Sections 4.1.1 and 4.1.2 look at point-to-point and collective MPI communications that are supported by PUPPET. In Section 4.1.3, we examine the unsupported routines out of the 48, and see why nine of them cannot be supported in the current form of PUPPET.

## 4.1.1 Point-to-Point Communication

There are eight types of send commands in MPI, half of which are non-blocking versions of the other four (denoted by prefixing an `I` (for *immediate*) to the name). *Buffered sends* (`MPI_Bsend`/`MPI_Ibsend`) and *synchronous sends* (`MPI_Ssend`/`MPI_Issend`) correspond to our notions: a buffered send copies the data into a buffer and returns, and a synchronous send waits for a matching receive to be posted. A *standard send* (`MPI_Send`/`MPI_Isend`) is typically a buffered send, although it may be synchronous, usually because of limited resources. Since there is no way to determine what method is used, we assume that standard sends are buffered. Finally, a *ready send* (`MPI_Rsend`/`MPI_Irsend`) can be used when a matching receive is guaranteed to be already posted. In this case, using a synchronous send does not significantly decrease performance, so one is typically used (with a few optimizations to the protocol); this is what we simulate.

`MPI_Recv` is a typical blocking receive-operation, and `MPI_Irecv` is the non-blocking version of it. `MPI_Wait` provides a wait feature, where a process explicitly asks to wait for a non-blocking operation to complete.

### 4.1.2 Collective Communication

There are several collective operations in MPI. Some have a variation (with a `v` postfix) where variable-length buffers are used. Groups are created to define the processes involved in a collective operation; these can be converted into a list of global process numbers using `MPI_Group_translate_ranks`.

`MPI_Barrier`, `MPI_Bcast`, `MPI_Scatter`, and `MPI_Gather` correspond to the barrier synchronization, multicast, scatter, and gather operations, respectively. `MPI_Reduce` (`MPI_Allreduce`) is the reduce operation where the result is left in one (all) process(es). `MPI_Allgather` forms a non-personalized all-to-all communication, where only $p$ messages are involved (where $p$ is the number of processes), and a personalized version (with $p^2$ involved messages) is given by `MPI_Alltoall`.

### 4.1.3 Unsupported Features

The above features are all supported by PUPPET. Some MPI facilities are not simulated: some we feel are not important and could be added by request, but others are in fact impossible to simulate. The former include `MPI_Waitall` (a repeated wait operation), `MPI_Probe` (a receive but without actually receiving the message), inactive requests (a way to avoid repeatedly passing the same parameters), `MPI_Sendrecv` (a combination of a send and a receive operation), `MPI_Reduce_scatter` (a combination of a reduce and a scatter operation), and `MPI_Scan` (a rare collective operation). The latter operations form the network-non-deterministic portions of MPI, which (as we noted in Section 2.1.2) are impossible to simulate with a loose coupling between the simulator and program. These features include `MPI_Test` and its variants, `MPI_Waitany`, `MPI_Waitsome`, `MPI_Iprobe`, `MPI_Cancel`, and `MPI_Test_cancelled`, which we now describe briefly.

`MPI_Test` returns a flag indicating if a pending request (returned by a non-blocking operation) has completed. Similarly, `MPI_Iprobe` checks if a matching message is available. The result of either routine usually completely changes the application's behavior; the result also likely varies significantly between the pseudo-parallel and simulated systems. Hence, loosely coupled simulation is inappropriate for these operations. `MPI_Waitany` and `MPI_Waitsome` allow non-deterministic selection between pending requests; the same argument applies.

`MPI_Cancel` attempts to cancel an operation, and `MPI_Test_cancelled` checks if the cancel succeeded. `MPI_Test_cancelled` is similar to `MPI_Test`, and so for similar reasons it is not supportable in the current simulation system. In addition, `MPI_Cancel` does not make sense without `MPI_Test_cancelled`.

The above features cannot be supported because of PUPPET's loose coupling. They would be entirely possible, however, in a tightly coupled version.

## 4.2 Logging Library

A loosely coupled simulation requires that an execution log be created. In this section, we see how MPI programs can be automatically instrumented. We also briefly examine other logging libraries that have been proposed.

MPI provides a *profiling interface* which defines how any instrumentation of MPI programs is performed. To do this, it places the following two main restrictions on an implementation of MPI:

1. An alternative library can be linked in where routine names start with `PMPI_` instead of `MPI_`.

2. A no-op routine `MPI_Pcontrol` is provided.

The idea is that the instrumenter can write a set of routines (whose names start with `MPI_`) that does any special logging or other operations before and after a call to the corresponding `PMPI_` routine. In addition, one can provide an `MPI_Pcontrol` operation to allow enabling, disabling, or otherwise controlling the instrumentation. If the standard MPI library is used instead, the `MPI_Pcontrol` operations will be ignored.

PUPPET's logging library creates a log file for each process. Each call to the relevant routines (listed in the previous section) induces an entry consisting of the routine name, properly encoded parameters, and the length of the previous computation block. Using `MPI_Pcontrol`, recording can be enabled or disabled, and buffers can be flushed or reset. Resetting buffers causes all log entries to be removed, except for the original `MPI_Init` entry (since it cannot be regenerated).

The format of the log files is simple and portable. It is designed so that it could also be used for future versions of MPI, or other message-passing systems such as PVM [37]. Once all of the processes exit, the logs can be combined into a single file via the UNIX `cat` utility, or PUPPET's `combinelogs` simplifying interface to `cat`.

## 4.2.1 Other Logging Libraries

The MPICH project [10] consists mostly of a portable MPI implementation, but also includes some utilities for use with any MPI implementation. In particular, MPE (MultiProcessing Environment) provides (among other things) a logging library for use with *upshot* [46], a performance-visualization tool. Similarly, the LAM project [13] has log-collection and display tools. Finally, Govindan et al. [39] have developed a logging library for use with the ParaGraph visualization tool [44].

All of these logging libraries append a real-time clock measure to each entry in the log. This is of course inappropriate for simulation of another parallel computer. Hence, none of these profiling systems are "accurate" in a pseudo-parallel system.

It would be possible to embed the entire simulator in PUPPET's logging library, yielding a tightly coupled simulation. In this case, we could (via the profiling interface) replace the `MPI_Wtime` routine (which normally returns the wall-clock time) to return the current simulation time. In this case, the above three logging libraries could be used so that a log compatible with a performance visualizer could be generated, since then the "real-time measure" would reflect the correct value for the simulated system. Unfortunately, the current MPI standard does not support multiple profiling interfaces; it may be possible by using the `wrappergen` utility of the MPICH project.

## 4.3 User Interface

The simulator takes either two or three command-line parameters. The first two are the names of a log file and a model file, respectively. The most interesting portion is the model file, which we look at in this section. Optionally, a PICL log file can be written, whose name is specified by the third parameter. The PICL log file is for use with the ParaGraph visualization tool (see Section 4.4).

Figure 4.1 gives a BNF-like grammar for model files. Basically, a model file consists of a sequence of *commands*, separated by newline characters. A command is either a comment (prefixed by a pound character) or an option. An option can either be global to the entire simulation, or local to a network for a particular kind of communication. In either case, the option that is latest in the file is the one used. For example, one could define the network for all communication types, and then define it for a particular one, "overriding" the previous declaration (but only for a single communication type).

There are currently two global options. The `barrier-size` option specifies the size of the message sent around to confirm that a barrier was reached, making a barrier simply a non-personalized all-to-all operation. In Section 3.4.2, we called

| | |
|---|---|
| model | $\rightarrow$ command `\n` ... |
| command | $\rightarrow$ globalopt $\mid$ networkopt [`for MPI_`. . . ...] $\mid$ `#` ...*comment*... |
| globalopt | $\rightarrow$ `barrier-size` $k$ $\mid$ `coll-sendtype` sendtype |
| sendtype | $\rightarrow$ `buffered` $\mid$ `synchronous` $\mid$ `nospace` |
| networkopt | $\rightarrow$ `network` net $\mid$ `map` $P_1$ ... $P_n$ $\mid$ `switching` model |
| | $\mid$ `link-speed` $i$ $t_1$ $\mid$ `packet-size` $p$ |
| net | $\rightarrow$ `complete` $n$ $\mid$ `tree` *arity level* $\mid$ `mesh` $k_1$ ... $k_m$ |
| | $\mid$ ...*etc*.... $\mid$ `custom` *file* |
| model | $\rightarrow$ `packet` $\mid$ `wormhole` $\mid$ `circuit` |

Figure 4.1: *BNF-like grammar for a model file, an input to the simulator.*

this value $k$.

`coll-sendtype` sets the type of the send operation when a collective communication is expanded into point-to-point operations. In addition to buffered and synchronous sends, a "no-space" send is possible. This is simply a send that completes once the message arrives. Such a send is likely what would be used when the standard send switches to "synchronous mode" because of resources becoming scarce; in the future, we might be able to determine when this happens, and in such a case switch to simulating a no-space send. In any case, we may use it for collective communication.

Each network, corresponding to a particular type of communication, requires that five options be provided. The first, obviously, is the network topology (specified by `network`). This can either be a built-in network, such as a completely-connected network of $n$ processors, a $k$-ary tree of depth $d$, or a $k_1 \times \cdots \times k_m$ mesh (which includes linear arrays and hypercubes), or it can be a custom user-specified network. In the latter case, the name of a network-specification file is given. This file lists those processors that each processor can send to directly; that is, it gives the incidence lists of the network graph (Section 3.2.1).

Second, a method for mapping processes to processors must be specified. This is particularly useful when doing multitasking, but can also be used to arbitrarily renumber built-in network topologies. There are two partially conflicting desires for the specification of this mapping:

1. It should be possible to arbitrarily assign processes to processors for a particular number of processes.
2. It should be possible to re-use a model file for any number of processes.

With a *wrap-mapping scheme* we assign process $i$ to processor $i \bmod P$, where $P$ is the number of processors (note that MPI processes are numbered 0 up). Obviously, this only satisfies condition 2. The opposite extreme is to list the processor assigned to each process. However, this only satisfies condition 1.

To satisfy both conditions, we chose a generalized wrap-mapping scheme, which essentially mixes the above two techniques. The user specifies a list $P_1, ..., P_n$ of processors. The first $n$ processes are placed on the respective processor in this list. In addition we wrap, so that the $(n+1)$st process is placed on processor $P_1$. If $P_i = i - 1$ for each $1 \leq i \leq n$ and $n$ is the number of processors, then this corresponds to a typical wrap-map. On the other hand, if $n$ is the number of processes, we have an arbitrary mapping.

Since the processor numbering and even the number of processors may vary across the various networks, we have no way of determining the correspondence between processors on the networks. To make any sense, each processor used for computation must be realized on each network; of course, the id of the processor may vary. Hence, we specify the mapping from processes to processors on each network. We also ensure that the mappings are identical aside from applying an invertible function to change the numbering. It would be possible to specify these functions instead of re-specifying the maps; we opted for the simpler approach.

The final three options specify the parameters to the network-link model (Section 3.2.2). They consist of the switching method, the link speed (initialization time per block and seconds per byte), and the size of a packet.

In addition to optionally generating a ParaGraph-compatible log file, the simulator gathers a few quantitative statistics. These include the time at which each process exited (and hence parallel execution time), the total amount of computation time (which equals the execution time if all processes were multitasking on a single processor, and hence we obtain a notion of speedup), the total amount of blocking (idle) time, and the utilization of each processor (plus an average utilization).

These statistics are useful for several reasons. First, ParaGraph does not yet support multitasking, and so some other output is needed when this feature is used. Second, creating a log file is typically very slow for reasonably long-running programs; it can be useful just to get an idea of where the program or model stands before going into detail. ParaGraph also does not support all of the statistics; for reporting on results, such as we do in Chapter 5, it is useful to have these numbers. The overhead of starting ParaGraph may also be unaffordable when several models are used, or if the user is on a slow network connection or even on a textual system where ParaGraph (which uses X-windows) can not be used.

## 4.4 ParaGraph

The simulator can generate an estimated real-time log file since we end up scheduling each event generated by the program. In particular, we chose the (new) format of PICL logs [80], since such logs can be used with the *ParaGraph* [44] parallel-program performance-visualization tool.

We chose ParaGraph because it is probably the most developed system of its kind. Heath et al. [45] state the design goals to be ease of understanding, ease of use, and portability; we feel that it has largely achieved these goals. It has many different displays, some of which are entirely unique from other visualization tools. Furthermore, it is very general; most other systems assume a particular parallel computer or a very specific architecture. ParaGraph is also probably the only free system under active development. We hope that it soon supports more processes than processors; if it does so, it will likely be the first system to do this.

We will see the usefulness of ParaGraph in Section 5.7.

## 4.5 Network Evaluator

A simple MPI program is provided in PUPPET to measure the performance of a network link. It assumes that the parallel system is completely isolated, so that wall-clock times are accurate. The basic idea is to use a ping-pong benchmark, in other words, measure the time for process A to send a message to process B and then send a message in the opposite direction. The simplest method is to do this for various messages sizes, and use a linear regression to find the link-speed parameters. However, we want the measuring to be robust, so that no or few outliers need to be removed. A particularly effective method for this is to pass messages of the same size several times, and take the median time to send a message. Outliers do not affect the median, which makes this method particularly robust.

We would also like to diminish the overhead of determining the current real-time clock value. To do this, one usually passes messages of the same size several times, but only measures the clock at the beginning and at the end. This length is then divided by the number of messages sent, resulting in the mean time to send a message. Unfortunately, a few outliers can significantly affect the mean.

Therefore, we adopt a hybrid approach, which we call the *mixed median-mean* approach. Essentially we use the mean approach several times for each message size, and then take the median of these values. This attempts to remove outliers while using the real-time clock relatively fewer times.

# Chapter 5

# Experiments and Results

In this chapter we validate the PUPPET simulation system. We evaluate its accuracy and show its usefulness for developing parallel applications. In Section 5.1, we overview the test applications that we use. Sections 5.2 and 5.3 examine the simulation accuracy and speed, respectively. We analyze the meaning of the speedup metric determined by the simulator in Section 5.4. Section 5.5 illustrates how the simulator allows evaluating load-balancing schemes. In Section 5.6, we see how one can experiment with different network topologies. Finally, Section 5.7 shows how we can understand programs and their performance with ParaGraph.

The reader may refer to Appendix A for the raw data that produced most of the plots found in this chapter.

## 5.1 Test Programs

We use three main programs throughout the evaluation. Two (which we developed ourselves) solve the sorting and Cholesky-factorization problems, respectively. The third is the Scalable BLAS library [15]. We describe these applications in the following three sections.

### 5.1.1 Merge-Sort

Sorting is a classic computer-science problem. A well-known $\Theta(n \log n)$ algorithm to solve it is *merge-sort*. The idea is to split the data in half, recursively sort each piece, and then merge them (a linear-time operation). We can generalize this to split into $k$ ($\geq 2$) pieces.

A parallel sorting algorithm is obtained by performing a single generalized merge-sort iteration with $k$ equal to the number of processors $P$. In other words, we split the data up into $P$ approximately-equal pieces, each lying on its respective processor. Any serial sorting algorithm can then be applied to each piece.

Finally comes a merging phase. For simplicity, we assume that $P = 2^t$ for some integer $t$. In this case, we can view merging as $t$ typical serial merging phases. All processors $p$ with odd identifiers (numbering them from zero up) send their messages to processors $p - 1$. Even-numbered processors then merge their data with the received data. The merging problem is then reduced to $P/2$ processors by discarding odd-numbered processors, and we continue as above.

As an example, we chose the bubble-sort $\Theta(n^2)$-time algorithm for local sorting. This has the advantage that small problems yield large execution times, making

```
For $j = 1$ to $n$,
    For $k = 1$ to $j - 1$,
        $\mathbf{L}_{*j} \leftarrow \mathbf{L}_{*j} - l_{jk}\mathbf{L}_{*k}$
    $l_{jj} \leftarrow \sqrt{l_{jj}}$
    $\mathbf{L}_{*j} \leftarrow \mathbf{L}_{*j}/l_{jj}$
```

(a)

```
For $k = 1$ to $n$,
    $l_{kk} \leftarrow \sqrt{l_{kk}}$
    $\mathbf{L}_{*k} \leftarrow \mathbf{L}_{*k}/l_{kk}$
    For $j = k + 1$ to $n$,
        $\mathbf{L}_{*j} \leftarrow \mathbf{L}_{*j} - l_{jk}\mathbf{L}_{*k}$
```

(b)

Figure 5.1: *The two column-oriented serial Cholesky-factorization algorithms, assuming that* $\mathbf{L}$ *is initially* $\mathbf{A}$. $\mathbf{L}_{*j}$ *denotes the* $j$*th column of the matrix* $\mathbf{L}$. *(a) Left-looking (jki) algorithm. (b) Right-looking (kji) algorithm.*

them less susceptible to slight deviations in measured running-time. It is also an interesting algorithm since using the conventional definition it obtains super-linear speedup.

### 5.1.2 Cholesky Factorization

A popular problem in numerical linear algebra is that of *Cholesky factorization.* The object is to factor a symmetric, positive-definite matrix into the form $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$, where $\mathbf{A}$ and $\mathbf{L}$ are both $n \times n$ and $\mathbf{L}$ is lower-triangular. We will assume that the matrix is dense, that is, there is no special zero-element structure; in this case, the problem is simple to parallelize.

Figure 5.1 gives the two main (equivalent) Cholesky-factorization algorithms. The $\mathbf{L}_{*j} \leftarrow \mathbf{L}_{*j} - l_{jk}\mathbf{L}_{*k}$ operation is called the *update* of column $j$ by column $k$, and the $l_{jj} \leftarrow \sqrt{l_{jj}}$, $\mathbf{L}_{*j} \leftarrow \mathbf{L}_{*j}/l_{jj}$ group operation is called the *normalization* of column $j$. The *left-looking* algorithm (Figure 5.1(a)) uses previous columns to update the current one, whereas the *right-looking* algorithm (Figure 5.1(b)) updates later columns once a column is completed. These algorithms can each be parallelized by

```
For $j = 1$ to $n$,
    If column $j$ is owned, then
        For each owned column $1 \leq k < j$,
            Update column $j$ by column $k$
        While column $j$ is not completely updated,
            Receive and apply a combined contribution for some column $j'$
        Normalize column $j$
    Else if at least one $1 \leq k < j$ is owned, then
        Calculate combined contribution $\mathbf{x}$ of all such $k$ for $j$
        Send $\mathbf{x}$ to the owner of column $j$
```

Figure 5.2: *The fan-in parallel Cholesky-factorization algorithm.*

evenly distributing the columns of **L**, yielding the fan-in and fan-out algorithms, respectively.

The *fan-in* algorithm (Figure 5.2) acts much like its serial counterpart. Each processor loops over $j$, the column to be updated, and decides its contribution to column $j$. All owned columns, say $k_1$, ..., $k_m$, with numbers less than $j$ are needed. If the processor owns column $j$, these updates are done locally; otherwise, they are collected via

$$\mathbf{x} = \sum_{i=1}^{m} l_{jk_i} \mathbf{L}_{*k_i}$$

(note that this computation can be done locally) and then $\mathbf{x}$ is sent to the owner of column $j$. Hence, the owner of $j$ must also receive these "combined contributions" until $j$ is completely updated. A small complication is that contributions may be received for some column $j' > j$ since the processors are not synchronized; in this case, column $j'$ is updated and more messages are awaited.

The *fan-out* algorithm [38] (Figure 5.3) was the first distributed-memory

```
If column 1 is owned, then
    Normalize column 1
    Send column 1 to all processors needing it
While the last owned column is not complete,
    Receive some column $k$
    For each owned column $k < j \leq n$,
        Update column $j$ by column $k$
        If column $j$ is completely updated, then
            Normalize column $j$
            Send column $j$ to all processors needing it
```

Figure 5.3: *The fan-out parallel Cholesky-factorization algorithm.*

Cholesky-factorization algorithm. Whenever a processor completes a column, the column is sent to all processors (including the local processor) that own a column affected by it. To make fan-out reduce to a serial algorithm when only one processor is available, one can change a "send column $k$ to myself" instruction into adding $k$ to a queue. When a receive is issued, the queue is first checked; if it is non-empty, column $k$ (which is stored locally) can be used.

It is worth noting that the communication patterns of fan-in and fan-out are extremely different. Fan-in combines messages wherever possible, greatly reducing the total message volume and the number of messages. On the other hand, fan-out significantly reduces the latency in waiting for messages, since messages are often sent before they are needed, and so they arrive (hopefully) before they are needed.

### 5.1.3 Scalable BLAS

Scalable BLAS (sB_BLAS) [15] is a parallel implementation of the level 3 BLAS (Basic Linear Algebra Subprograms). It is based on the partitioning of matrices into *blocks* (hence the abbreviation sB) in the SUMMA (Scalable Universal Matrix Multiplication Algorithm) [76]. The SUMMA work was extended to all of the matrix-matrix operations in the (serial) level 3 BLAS. These include multiplication of general, triangular, and symmetric matrices; symmetric rank-$K$ and rank-$2K$ updates (forming $\mathbf{A} \cdot \mathbf{A}^T$ and $\mathbf{A} \cdot \mathbf{B}^T + \mathbf{B} \cdot \mathbf{A}^T$, respectively); and back-substitution (solving $\mathbf{L} \cdot \mathbf{x} = \mathbf{y}$ for $\mathbf{x}$) with multiple right-hand sides.

The algorithms are proven to be very scalable under a reasonable model. That is, if $P$ is proportional to the problem size (the order of the matrix squared), then the efficiency (speedup over the number of processors) is $1/\left(1 + O\left(\log p\right)\right)$. Experimental results also show that this theoretical bound is realized.

We will not cover the algorithms in detail. The basic MPI operations that are used include collective communication (multicasting and reduction), synchronous communication, and non-blocking communication. Hence, these programs offer extensive tests of most of the simulator's features.

## 5.2 Simulation Accuracy

In this section we evaluate the accuracy of simulation for the various applications in many problem sizes. To do this, we run the parallel programs on a real parallel system, and compare to the simulation output for a pseudo-parallel run using a single node of the system. The measure that we compare is total parallel execution time. Throughout this section, the number of processes equals the number of processors.

We used two parallel systems in these experiments. The first is a network of eight identical IBM RISC System/6000 workstations connected via a 1.25 Mbyte/sec. Ethernet. This was provided by the Shoshin lab at the University of Waterloo. The second is Dalhousie University's IBM SP2 with four "thick" nodes connected by a 35.5 Mbyte/sec. switch.

Obviously, the SP2 provides much higher performance than the Ethernet LAN; it has superior processing power and networking. This has the interesting effect that "granularity" has a different interpretation. That is, the same program with the same problem size has much less computation between each communication, if we use an absolute measure (e.g., seconds) instead of one relative to the network speed. We will see how this is important in Section 5.2.2 since we use CPU time (which is measured in centiseconds) when logging pseudo-parallel runs.

### 5.2.1 Ethernet

We used the network evaluator (Section 4.5) to determine the performance of message-passing via the MPICH [10] implementation of MPI. The raw output is given in Figure 5.4. Currently, we manually remove outliers (yielding Figure 5.5) and do a least-squares fit. The resulting model is approximately $1449.848 + 1.008644k$ microseconds for a $k$-byte message. As one can see from Figure 5.5, the linear model is accurate; obviously, this is a necessary condition for

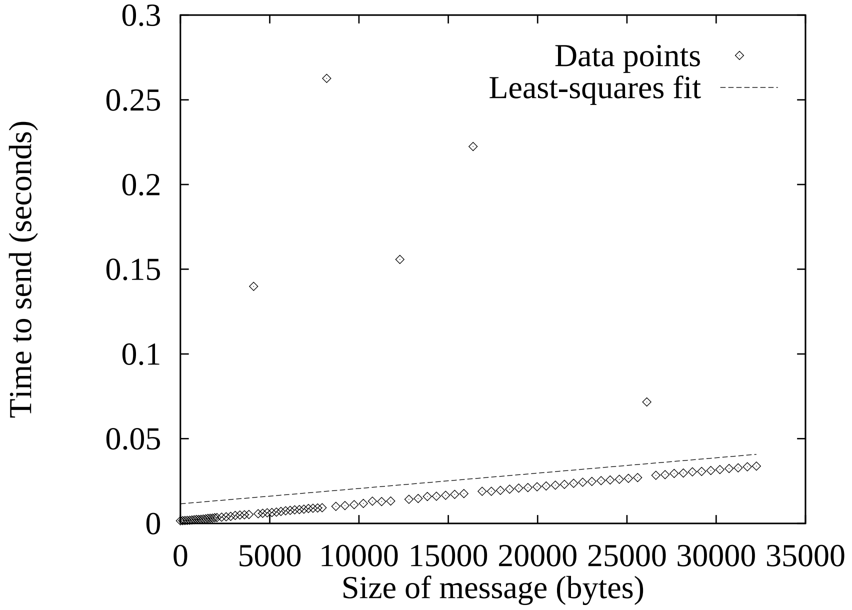


Figure 5.4: *Raw data from the network evaluator applied to the Ethernet network, with a least-squares fit. The outliers cause this fit to be inaccurate.*

an accurate simulation. Since we ignore network traffic, the Ethernet is simulated as a completely-connected network with links of this speed.

Figures 5.6 and 5.7 summarize simulation accuracy for various applications and problem sizes. Overall, we can see that accuracy is quite impressive. Error is consistently below 10%, and often below 5%. In the rest of this section we suggest reasons for the observed accuracy pattern.

There are two basic points where the simulation may not be accurate: simulating the computation and simulating the network. We use CPU time to measure the length of computation blocks, which is only accurate to centiseconds. Second, we do not simulate network congestion (from the above results we are convinced that the network-link model is not at fault). CPU time is more accurate when computation

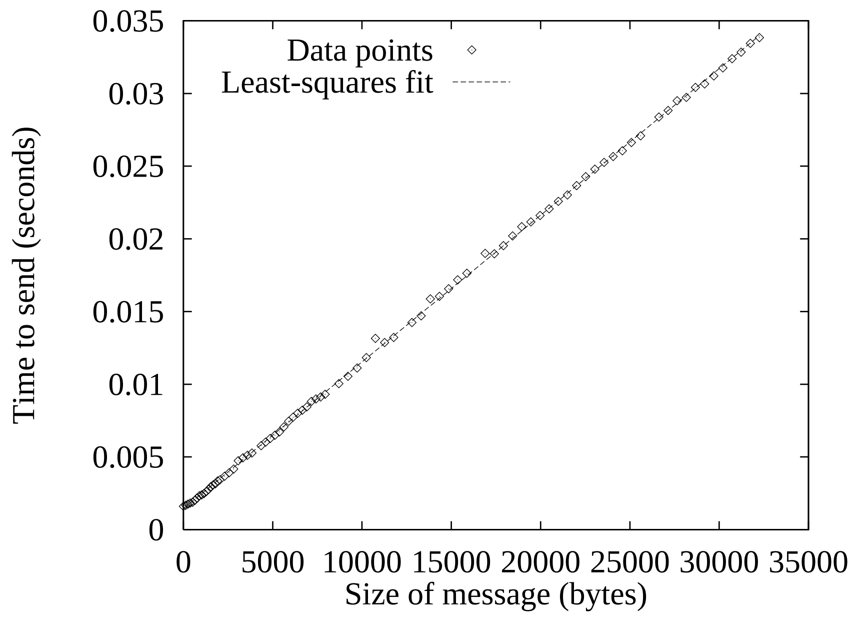


Figure 5.5: *Data from Figure 5.4 with outliers removed. This allows the least-squares fit to be accurate.*

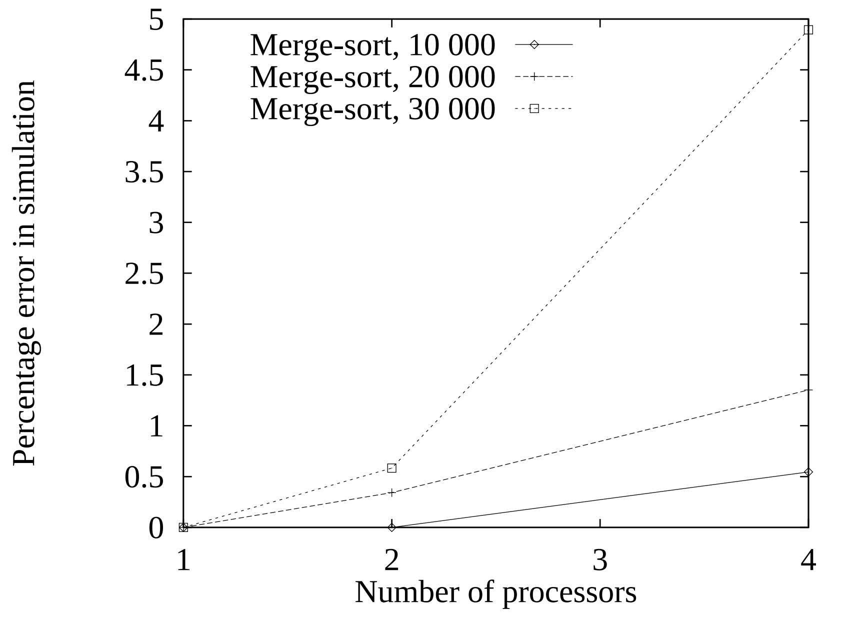


Figure 5.6: *Simulation accuracy for the merge-sort program on the Ethernet. The sign of the error has no predictable trend.*

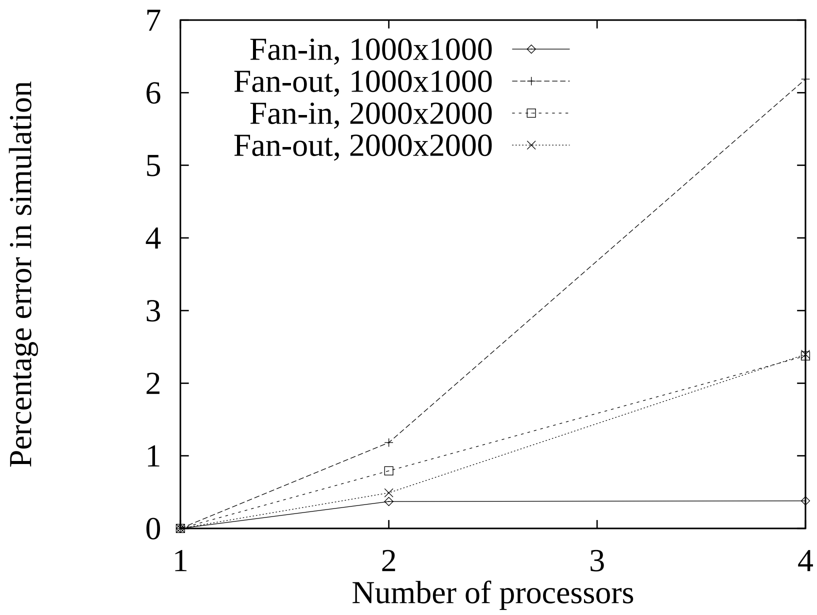


Figure 5.7: *Simulation accuracy for the Cholesky-factorization programs (fan-in and fan-out) on the Ethernet. The simulation almost always under-predicts execution time.*

blocks are larger, and the network simulation is more accurate when there is less network traffic.

Let us first examine the merge-sort results (Figure 5.6). Here there are two simple trends: (1) more processors implies less accuracy, and (2) a larger problem implies less accuracy. It is likely that both are caused mainly by network traffic. Using more processors causes more messages, and using larger problems causes larger messages. Using more processors also causes some inaccuracy in CPU time, but this is likely minimal or else bigger problems (yielding larger computation blocks) would greatly improve accuracy.

An important point is that the Ethernet is a bus, but we simulate a completely-connected network. Only one message can occupy the bus at a time. For small-enough messages and few conflicts (that is, few processors), the difference is not noticeable. For the largest problem and four processors, it is likely that this serializing of messages (which we do not simulate) is causing much of the 5% inaccuracy.

Let us now turn to the Cholesky-factorization algorithms (Figure 5.7), where the interaction is more complicated. Here we run into both inaccuracy problems. Fan-out has much more communication than fan-in. In addition, fan-out has some very small computation blocks in between message-sends. For a $1000 \times 1000$ matrix, fan-out is not simulated as well as fan-in, likely for both reasons. When we move to $2000 \times 2000$, fan-out becomes better for two processors and returns to (slightly) worse than fan-in at four processors. This is likely because traffic is not an issue when only two processors are involved, and hence the benefit in CPU-time accuracy is high; since CPU-time accuracy is more important in fan-out, it ends up having more accuracy in this case.

We were unable to run further tests (for example, with eight processors) because of problems in using the Ethernet. In particular, we did not gather any results for the Scalable BLAS. In this case, however, more results are likely unnecessary, since the sorting and Cholesky-factorization applications are accurately simulated.

### 5.2.2 SP2

An advantage of the SP2 is that an affine model of network link performance is published. The model states that the time to send a message of length $k$ is approximately $39 + 0.026864k$ milliseconds. Thus, we did not need to run the network evaluator.

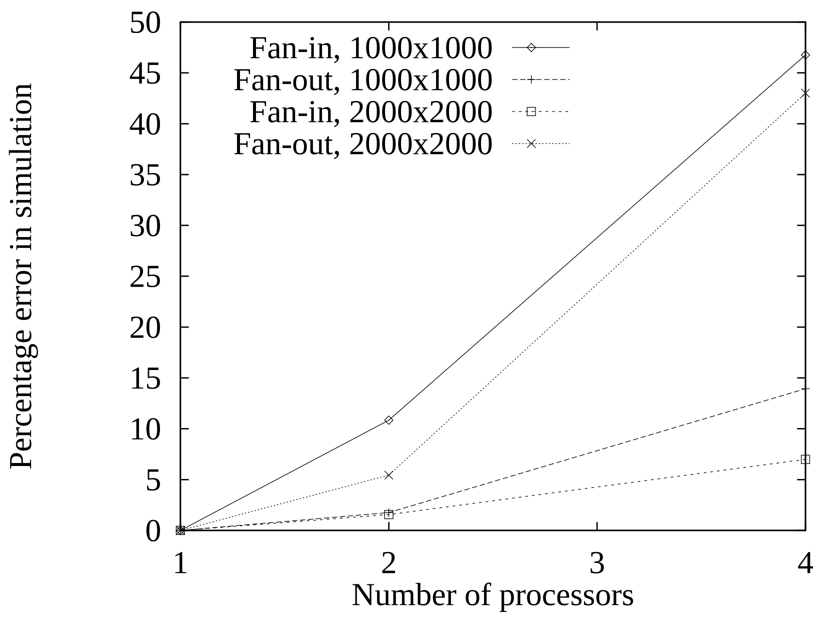


Figure 5.8: *Simulation accuracy for the Cholesky-factorization programs (fan-in and fan-out) on the SP2. The simulator consistently over-predicts for fan-in, and under-predicts for fan-out.*

As mentioned earlier, the SP2 has much faster processors. Because of this, the merge-sort and Cholesky-factorization problems of the above sizes are trivial to solve. The merge-sort algorithm takes so little time that it was impossible to measure accuracy because of slight variance in runs. Cholesky factorization was not as bad, but it still does not offer a coarse-enough grain with problems that fit in main memory. Hence, CPU times are very inaccurate and the results (Figure 5.8) are poor.

For the above reasons we need larger (and therefore coarser-grain) applications that do not require as much memory. This is why we chose the Scalable BLAS library. We ran the four main subroutines each with matrices of size $800 \times 800$, $1600 \times 1600$, and $2400 \times 2400$. The results are given in Figures 5.9, 5.10, and 5.11,

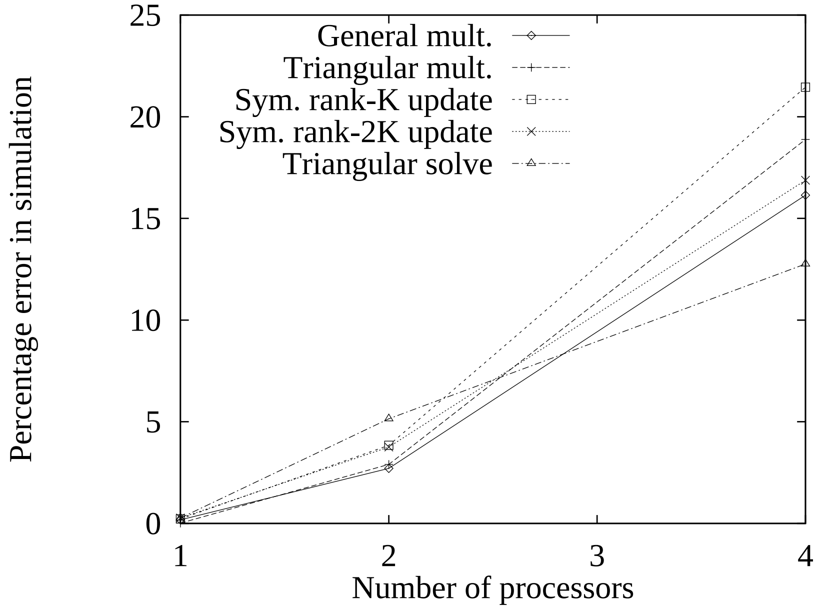


Figure 5.9: *Simulation accuracy for the Scalable BLAS with a* $800 \times 800$ *matrix on the SP2. For two and four processors, the simulator consistently over-predicts the execution time.*

respectively.

It is clear that the $800 \times 800$ problem has too fine a grain. This can also be derived from the fact that the problem is too small, as it takes less than three seconds to run in each case. This leads to the inaccuracies shown in Figure 5.9, which is only two times better than the Cholesky-factorization results in the worst case. Fortunately, it was easy to use larger problems.

The $1600 \times 1600$ results (Figure 5.10) are impressive, consistently staying below 6% relative error. Hence, this problem size yields a satisfactory granularity, and CPU time is good at approximating the length of computation blocks[1]. As ex-

[1]Note however that it is not perfect, since 1-processor simulation has some inaccuracy, because we sum the (approximate) lengths of computation blocks instead of taking total execution time.

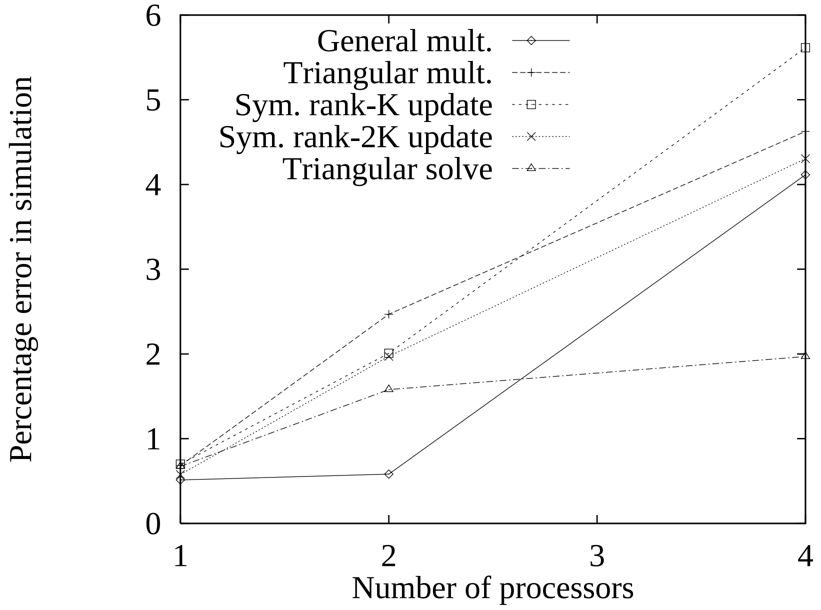


Figure 5.10: *Simulation accuracy for the Scalable BLAS with a* $1600 \times 1600$ *matrix on the SP2. For two and four processors, the simulator consistently over-predicts the execution time.*

pected, the accuracy goes down with more processors, since then the computation blocks shrink significantly.

Surprisingly, some of the results get worse when we switch to a $2400 \times 2400$ matrix. One would expect that, since the computation blocks get larger, CPU time is more accurate, and hence the simulation is more accurate. The reason for partial failure (sometimes reaching 8% error) is not the same as with the Ethernet, where network traffic was incorrectly simulated. With the SP2 switch, there is no network traffic.

Note that there are essentially two sets of lines. The top group (with two lines) reaches up to the 8% error, whereas the bottom group (with three lines) only reaches up to 3% error, which is much better than the 6% obtained with the $1600 \times 1600$

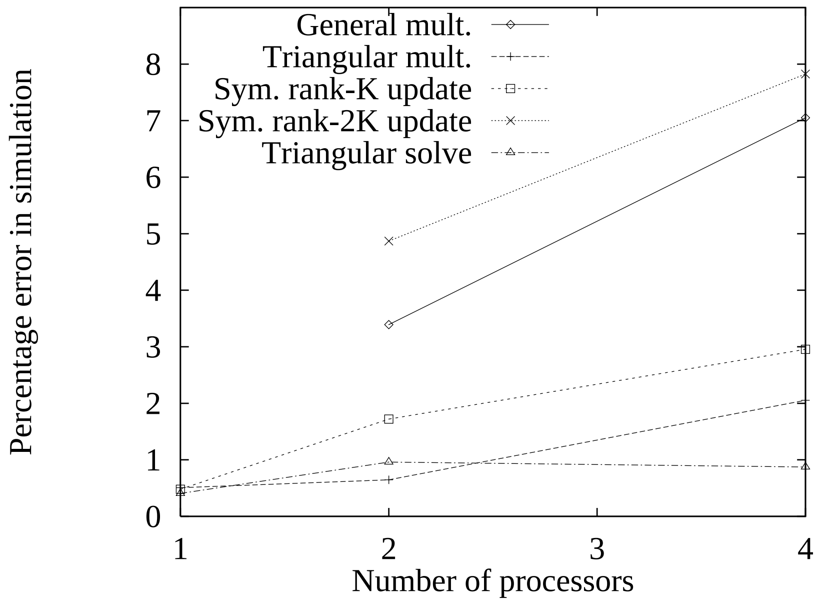


Figure 5.11: *Simulation accuracy for the Scalable BLAS with a* $2400 \times 2400$ *matrix on the SP2. For two and four processors, the simulator consistently over-predicts the execution time.*

problem. Thus, there is only a problem with the top two lines, which correspond to general matrix-matrix multiplication and symmetric rank-2K updates.

After considerable effort we found the reason that we obtain lower accuracy for these two problems is memory limitations. The two problems used significantly more (approximately double) memory than the others. General matrix-matrix multiplication does not have a symmetric problem, and the symmetric rank-2K update has another matrix involved. We noticed the same difficulty when 1-processor runs were attempted with a $2400 \times 2400$ matrix and the program attempted to allocate blocks that were larger than the maximum allowed (hence the missing data points in Figure 5.11). For simulating two and four processors (where the programs at least run), the single node that was used in pseudo-parallel swapped to disk.

One would hope that swapping would not affect the user CPU time measure, which is what the logging library uses. Unfortunately, experiments on the SP2 show that 30% error in CPU times can be observed in conjunction with a fair amount of swapping. This was determined by running a program whose execution time is proportional to the amount of memory used, so that the actual amount of computation is predetermined by running it with multiple problem sizes. Because of this error, computation blocks in the two Scalable BLAS problems were incorrectly measured, yielding the observed inaccuracy.

## 5.3 Speed of Simulation

A primary goal in the development of PUPPET was an extremely fast simulation. We developed an approximate but highly efficient network model by ignoring network traffic. This also applies to collective operations, greatly improving the performance of simulating them.

As Figures 5.12 and 5.13 show, the goal was reached. The delay involved in simulating appears to be linear in the number of log entries. Furthermore, the constant involved is very low. More than 10,000 entries are needed before the simulation takes five seconds. All of the Scalable BLAS programs, which made up to 3,376 entries, took less than a third of a second.

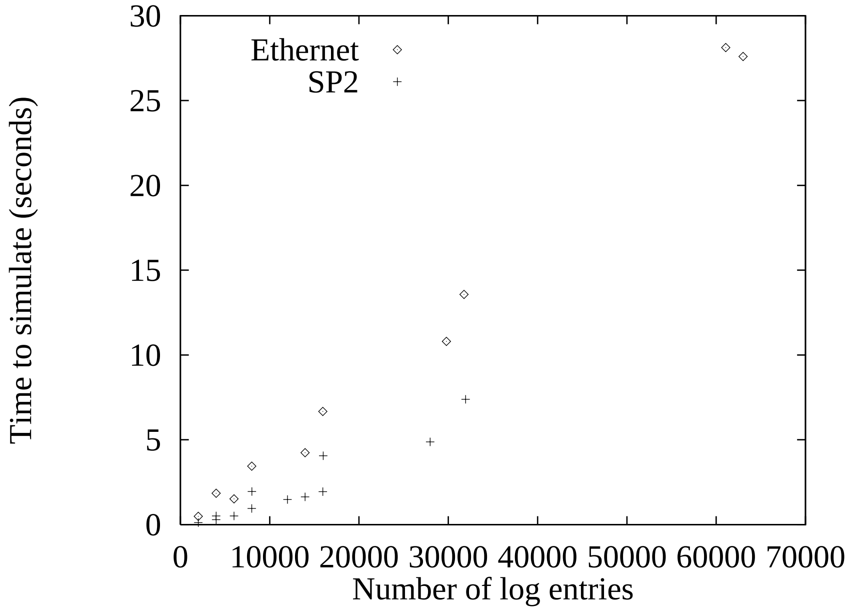


Figure 5.12: *Time for a node of the SP2 and a RISC/6000 to simulate Cholesky-factorization executions on the SP2 and the Ethernet, respectively.*

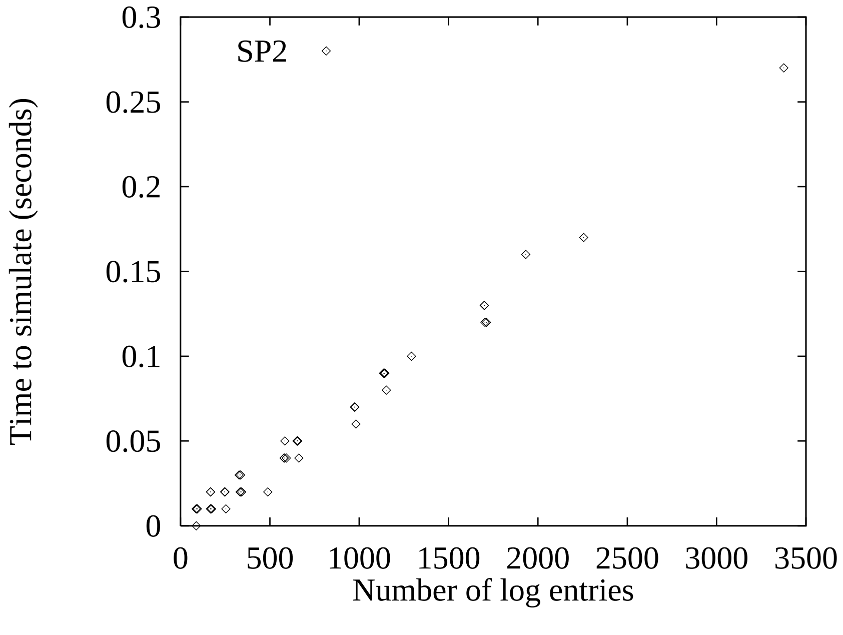


Figure 5.13: *Time for a node of the SP2 to simulate various Scalable BLAS programs.*

## 5.4 Scaled Speedup

Given just a parallel execution, the simulator estimates notions of *serial execution time* and *speedup*. These are not the traditional interpretations of Amdahl [2]; obviously, if a serial execution is not given, typical serial time cannot be calculated. Instead, serial execution time represents the time it would take to run the parallel program on a single processor by multitasking.

As one would expect, speedup is this measure divided by estimated parallel execution time. Gustafson, Montry, and Benner [43] were the first to use this metric, and called it *scaled speedup*. It compares the advantage of using multiple processors (parallelism) to the advantage of using a single processor (fast communication).

Scaled speedup is a good measure for evaluating parallel-program performance. It accounts for implicit change in problem complexity when the number of processors changes. Thus, it is never super-linear.

As an example, let us examine the parallel merge-sort application; recall that it is based on an $O(n^2)$ serial algorithm and hence obtains super-linear speedup (Figure 5.14). Assuming a perfect parallelization of bubble sort, one would expect speedups of 4, 16, 64, ... for 2, 4, 8, ... processors. To convert this to something that makes more sense, we could take the square-root of the speedups (Figure 5.15). Of course, this doesn't take into account the merging phase, which consists of $O(n \log n)$ computation and $O(\log n)$ message-send phases.

As one would hope, scaled speedup (Figure 5.16) is slightly lower than the square-root of typical speedup, since the merge phase makes the algorithm somewhat less efficient. It essentially *scales* speedup appropriately, using a function like

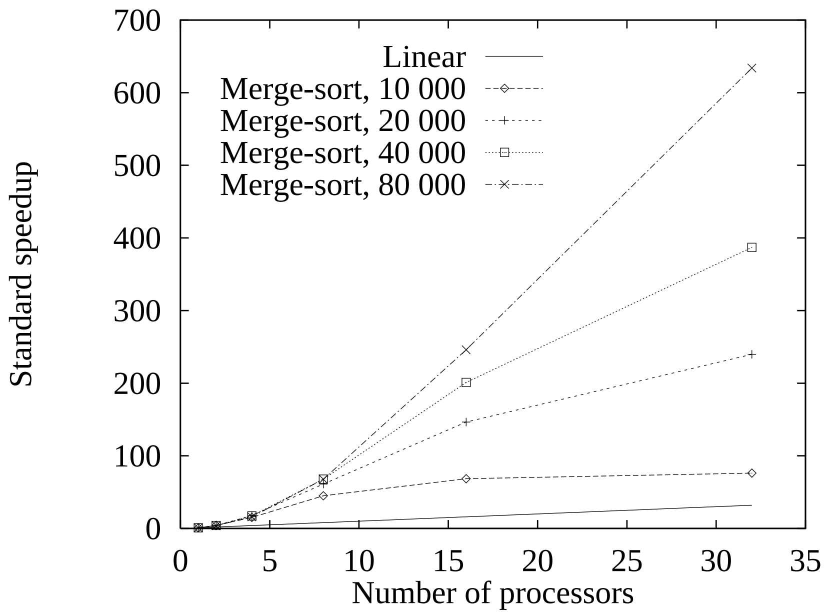


Figure 5.14: *The typical notion of speedup (time for serial algorithm over time for parallel algorithm) for the bubble-based merge-sort algorithm on the simulated Ethernet.*

square-root. However, it is very general, and requires no insight into the underlying algorithm (e.g., $O(n^2)$ complexity).

## 5.5 Load Balancing

One of the unique features of PUPPET is its support for multitasking, that is, more processes can be simulated than processors. The user can specify an arbitrary process-to-processor mapping. In particular, this can be used to evaluate various load-balancing schemes.

In this section we examine a simple load-balancing method, as applied to the fan-in algorithm run on the Ethernet. To make the problem interesting, we choose

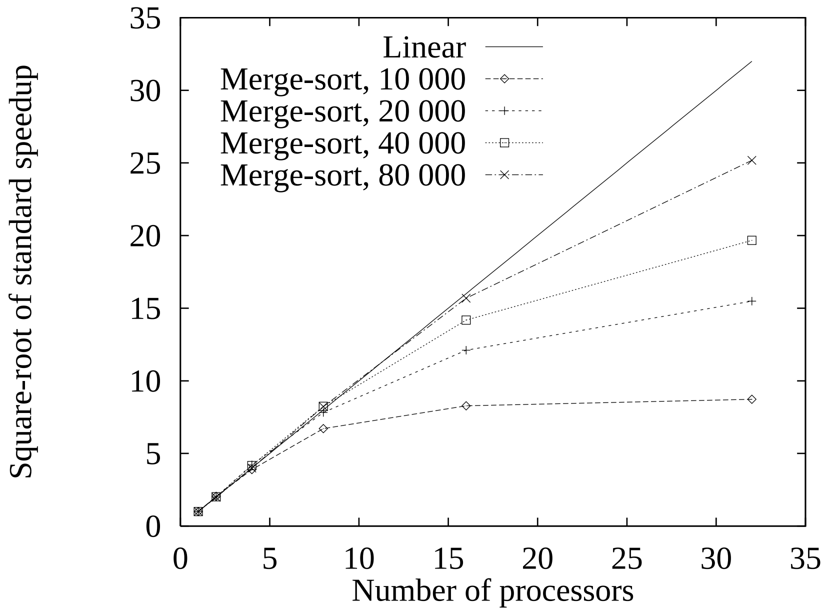


Figure 5.15: *The square-root of typical speedup for the bubble-based merge-sort algorithm on the simulated Ethernet. This takes into account the $O(n^2)$ complexity of bubble-sort, but ignores the merging phase.*

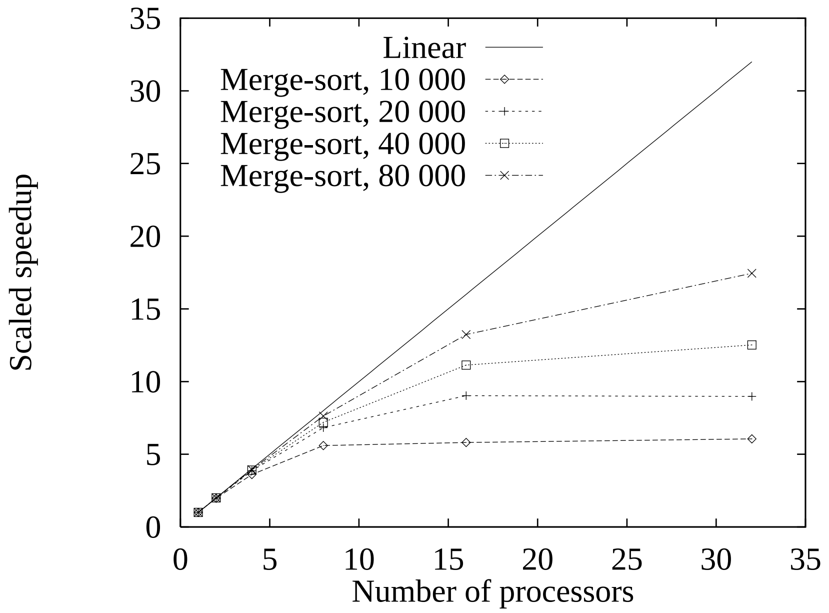


Figure 5.16: *Scaled speedup for the bubble-based merge-sort algorithm on the simulated Ethernet. This is an effective evaluation of the algorithm's performance.*

$P$ (the number of processors) to not evenly divide $p$ (the number of processes). Namely, $P$ is 9 and $p$ is 32.

The balancing algorithm that we chose to evaluate starts with a random mapping. This mapping is then evaluated via the simulator; since PUPPET is loosely coupled, the program only has to be run once. We move one process from the best-utilized processor to the least-utilized processor. We then continue by simulating and moving another process, until we cycle back to a mapping that we have seen before. That is, we try to balance processor utilization.

Figure 5.17 shows the evolution of three attempts, each starting at different randomly-chosen mapping. We use parallel execution time as an indication of how good the mapping is. One can see that the effectiveness of the load-balancing algorithm depends greatly on the starting point.

To illustrate the improvement over a random mapping, we plotted the utilizations at iterations zero (Figure 5.18) and nine (Figure 5.19) for attempt three, which represent the worst and best mappings for this attempt. The load-balancing scheme seems to be doing well at distributing the work given enough attempts.

## 5.6 Different Topologies

A crucial advantage of simulation is that one can evaluate systems that one does not have access to, or that do not even exist. In this section we consider a simple example of this; there are many situations where it is useful.

Suppose we are considering buying a small LAN for use with a special application that has characteristics similar to merge-sort. There are two options available

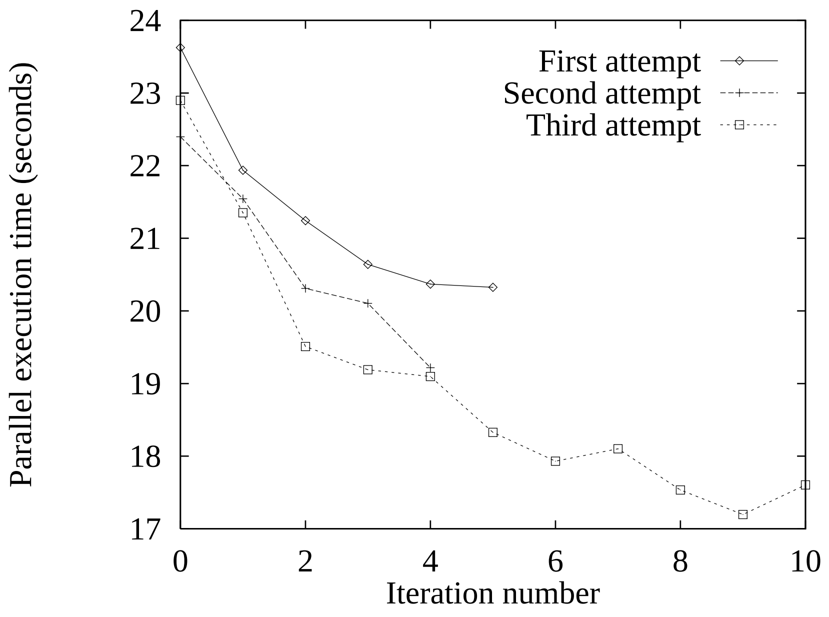


Figure 5.17: *Iteration-by-iteration view of the simple load-balancing scheme for fan-in for three attempts.*

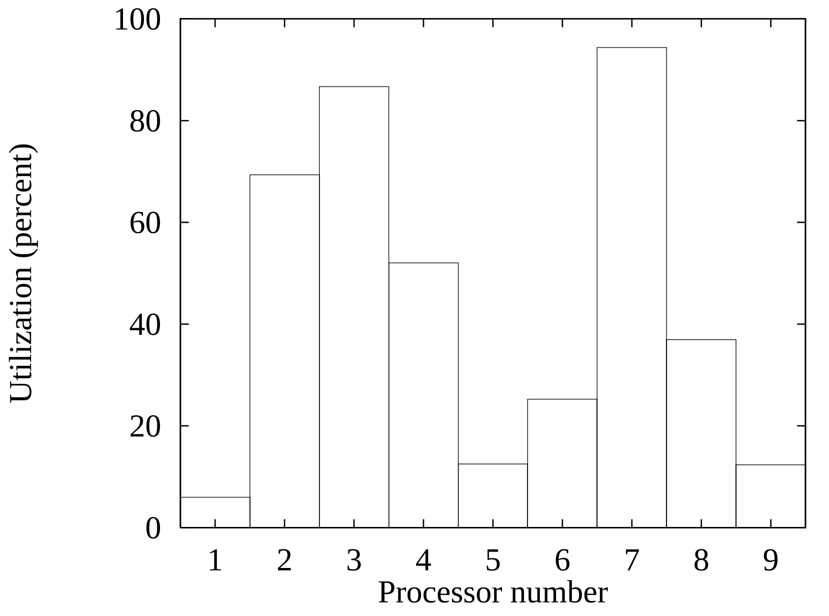


Figure 5.18: *Utilization of processors for initial random mapping of attempt three.*

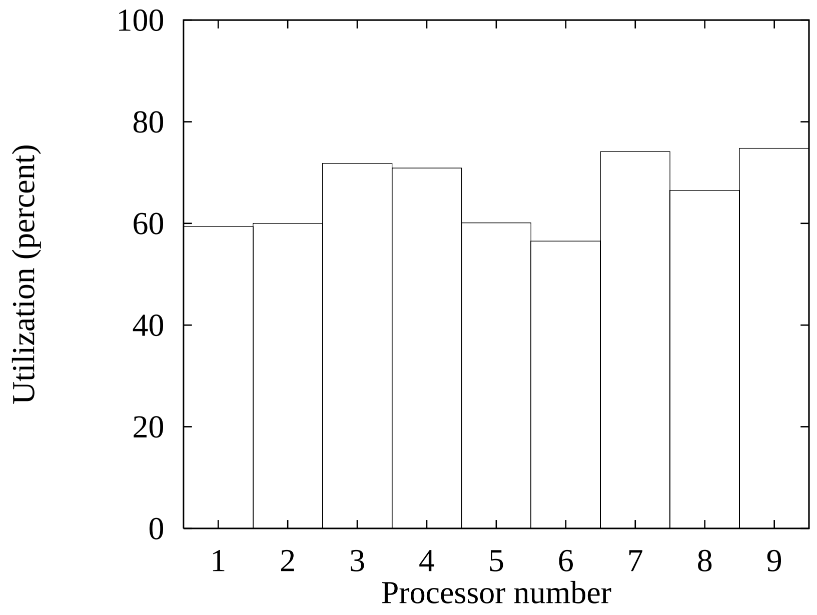


Figure 5.19: *Utilization of processors for best mapping found in attempt three.*

to us from our vendor, who provides network links of speed similar to the Ethernet. Either we obtain a hypercube topology or a ring topology, the former being more expensive. (Note that a hypercube topology is optimal for merge-sort, since the communication pattern is a subset of a hypercube; see Section 5.7.1.) We would like to evaluate if it is worth the extra money for the extra performance.

Figure 5.20 shows results for sorting a 10,000-integer array on the two possible (simulated) computers. It is clear that 16 processors (yielding a scaled speedup of 11) are not efficiently used. For 8 processors and less, there is a small difference between the two systems (for this application). Hence, we would likely opt for the cheaper ring network. If we are willing to buy 16 processors, we would also likely buy a hypercube network, or else it would hardly be worth the eight extra processors.

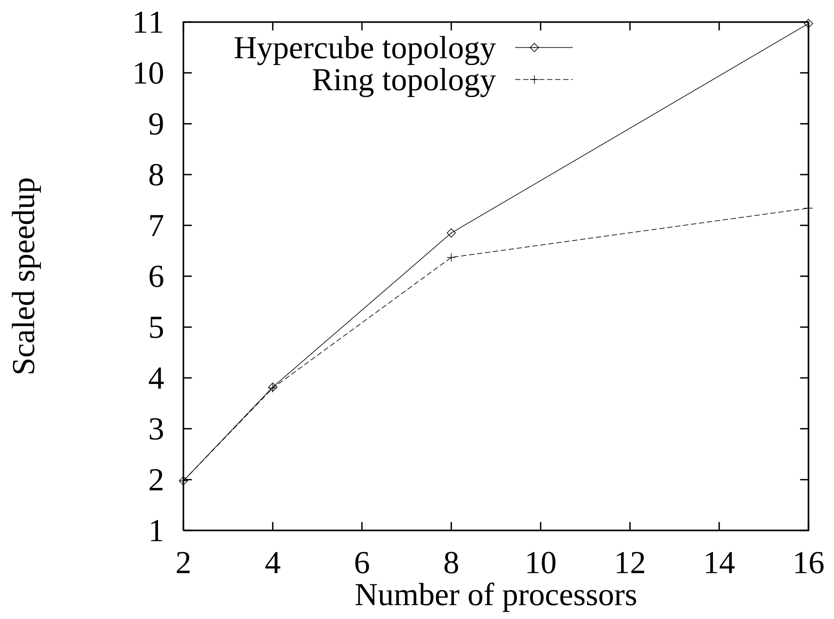


Figure 5.20: *Scaled speedup for merge-sort program for two different topologies.*

As one can see, it is easy to estimate performance for a ring (or other topology) supercomputer using the proposed simulator. In addition, multiple topologies can be tested and compared, without rerunning the programs. For example, one could compare a program's performance on a Cray T3D and an Intel iWarp, which have 3-D torus and hypercube topologies, respectively, without access to these machines, assuming that the network model is sufficient, and the network-link model parameters are known. (Without augmentation or instruction-level simulation, one would also need access to processors equivalent to those on the supercomputers.)

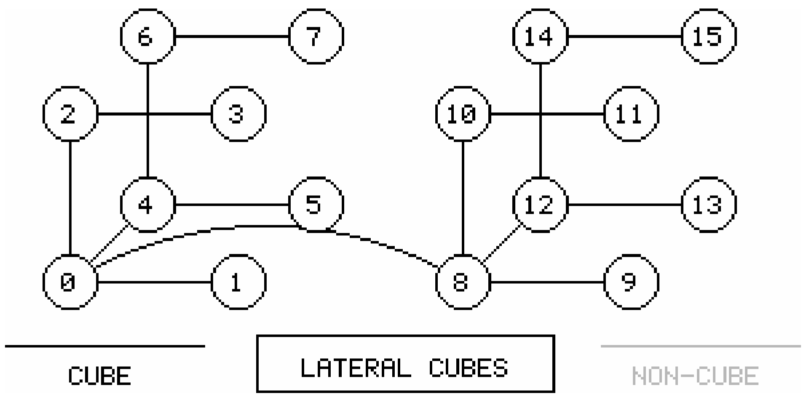


Figure 5.21: *The hypercube display after animation for the merge-sort program.*

## 5.7 ParaGraph

Recall that the simulator can generate PICL logs for use with the ParaGraph [44] parallel-program performance-visualization tool. In this section we demonstrate the usefulness of ParaGraph for performance- and program-understanding. In particular, we show snapshots of ParaGraph using the merge-sort and Cholesky-factorization algorithms.

### 5.7.1 Merge-Sort

Let us first consider the merge-sort program.

We can verify that the communication pattern for merge-sort is in fact a subset of a hypercube using a built-in ParaGraph feature. The *hypercube display* animates the communications in the program, and at the end (Figure 5.21) shows all links that were used. Those that are outside a hypercube topology are colored gray. Since all of the links are colored black, a hypercube is optimal for merge-sort.

One can also visualize the estimated real-time log with horizontal *trace lines*

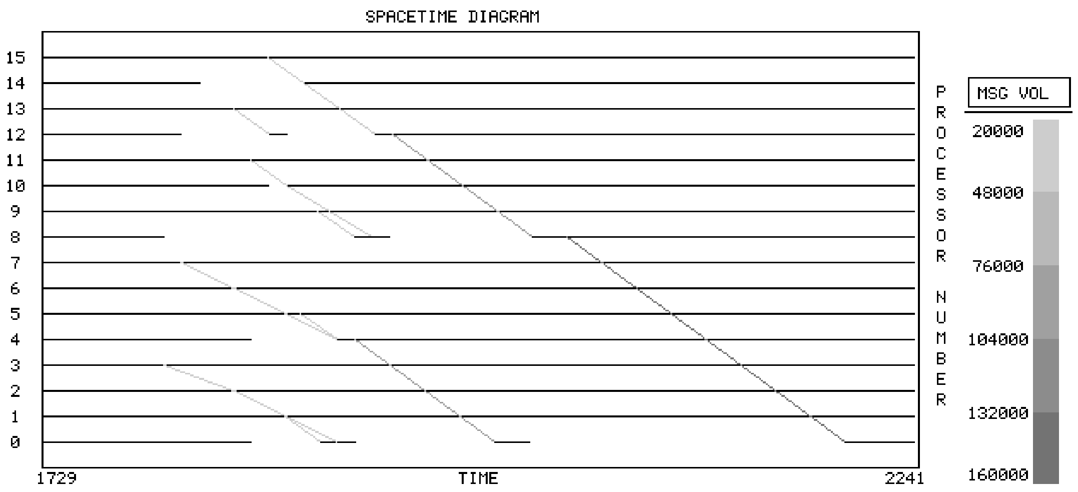


Figure 5.22: *The tail-end of the space-time diagram for the merge-sort program.*

(representing processors) and other lines denoting messages traveling between processors. This is called a *space-time diagram* (Figure 5.22). This view is useful to get an overall idea of what the parallel program is doing. At times it can even be used to spot bugs in the program by noting failed assertions such as processor 0 sending to processor 2 before processor 1 when it is supposed to communicate with processors in ascending order.

## 5.7.2 Cholesky Factorization

Let us now turn to the Cholesky-factorization problems, fan-in and fan-out. It is not clear exactly how the two compare (for example, we may ask which is faster for a $1000 \times 1000$ problem). In this section we attempt to answer such questions by understanding the performance of the two programs.

First we look at the space-time diagrams for each (Figures 5.23 and 5.24). We found it quite interesting to see this pattern for an algorithm which we have long

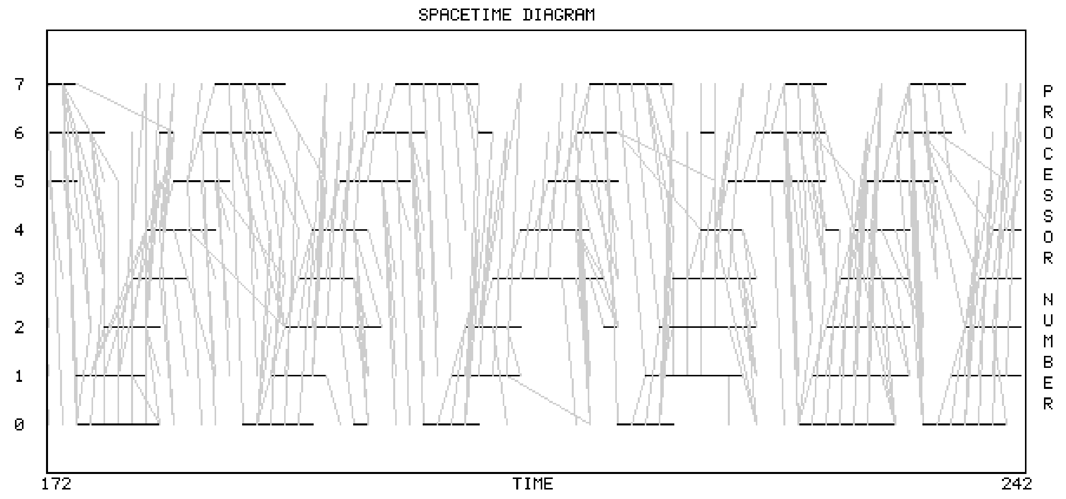


Figure 5.23: *A window of the space-time diagram for the fan-in program.*

worked with, but have only now "understood." For example, the fan-in algorithm has a kind of cascaded pattern of computation blocks; this suggests that overlapping computation and communication may be useful. One can also see that, as we guessed from examining the algorithms, fan-out has much more communication but less idle time (smaller gaps in the trace lines).

This is made clearer by the phase portraits of the two executions (Figure 5.25). A *phase portrait* is a history of processor utilization ($x$ axis) and network utilization ($y$ axis) summaries. The various points are connected by line segments to show the order. One can see that fan-out typically has higher values in both dimensions, signifying that there is less idle time but that there is more network traffic.

As a final example, we examine the message queue on each processor (Figures 5.26 and 5.27). This view is most useful as an animation, which is difficult to show here. We give snapshots to illustrate the general idea. Fan-in tends to have concentrated "waves" of queue congestion that move in increasing order among the

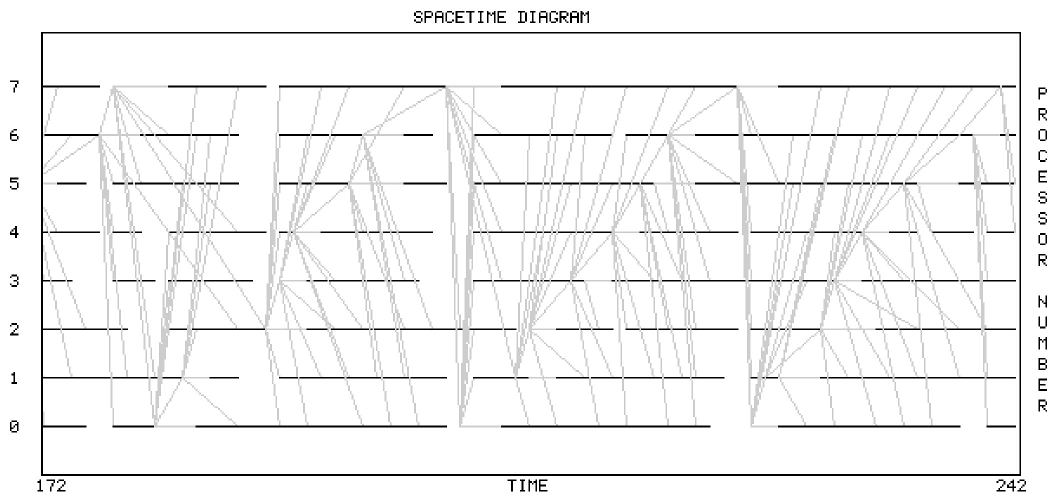
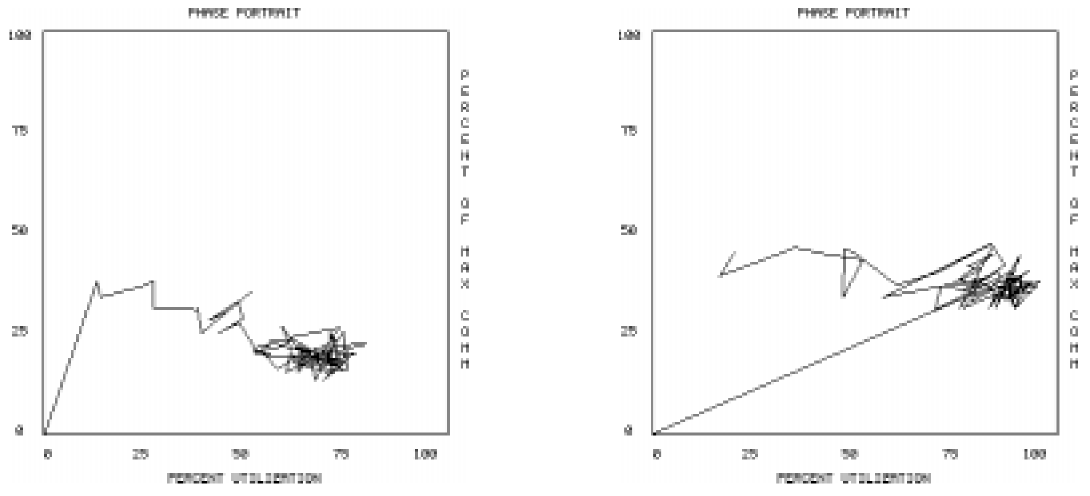


Figure 5.24: *A window of the space-time diagram for the fan-out program.*



(a) (b)

Figure 5.25: *Phase portraits for (a) fan-in and (b) fan-out.*

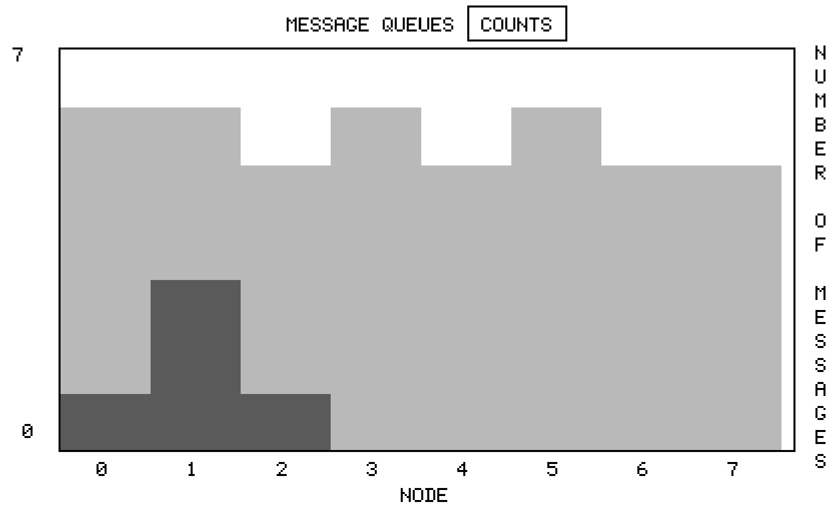


Figure 5.26: *A snapshot of the queue lengths (black) at each processor for fan-in, including the maximum sizes reached so far (gray).*

processors; in Figure 5.26 the wave is currently centered around processor 1. On the other hand, fan-out tends to have significant queue lengths at every processor, although it reaches a much lower maximum than fan-in. Again, we have obtained more insight into the programs than we had before.

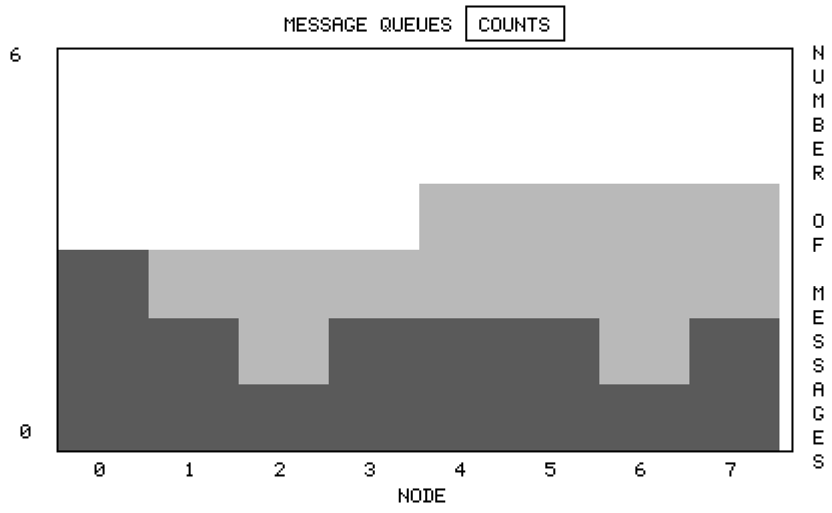


Figure 5.27: *A snapshot of the queue lengths (black) at each processor for fan-out, including the maximum sizes reached so far (gray).*

# Chapter 6

# Conclusion

In this thesis, we have shown that parallel-computer simulation is a powerful technique. Users can develop their programs in a pseudo-parallel environment and estimate the performance of a real parallel computer. This allows avoiding use of supercomputing resources when it is not necessary, for example, during the development phase of an application or for learning to program in parallel. Simulation also allows one to evaluate systems that do not yet exist, which is useful when considering buying a parallel computer.

This thesis focused on the development of PUPPET (Performance Under a Pseudo-Parallel EnvironmenT), a system for simulating MPI-based distributed-memory parallel computers. In compromising the accuracy of network simulation somewhat, it achieves incredible performance. The simulation has a loose coupling with the program, so that a program can be evaluated on many architectures by running the program only once.

## 6.1 New Features

PUPPET has several features that have not been provided (as far as we know) by any other simulation system. One primary difference is that it automatically instruments the user's program, written in MPI. MPI is one of the most popular programming languages based on the message-passing paradigm. Most simulation systems require that the user write in a special or uncommon language, making it difficult to later port and use the program on a supercomputer.

The overall goal of a simulation is to schedule each program-generated event to determine the program's performance, essentially making an estimated real-time log of the execution. PUPPET is the first simulator that actually generates a real-time log in a popular format that can be used by other programs, instead of taking on the large research area of performance visualization when there is already significant progress. In particular, we chose the log format supported by ParaGraph, since it is likely the most-developed public-domain visualization tool.

This thesis proposes a new model of direct networks, including details on how to support both point-to-point and collective operations. The model ignores network traffic, which is particularly effective when traffic is not an issue (for example, in an IBM SP2 which uses a crossbar switch). PUPPET adopts this scheme for network simulation since it offers great performance benefits.

Because of this network model, PUPPET supports the collective communication constructs of MPI, which is also unique. In fact, PUPPET supports all of MPI's deterministic features, including non-blocking communication. Three kinds of send operations are supported, buffered, synchronous, and no-space (where the send completes once the message arrives), the latter two of which we believe to be entirely new.

Another feature that has not been examined before is multitasking. PUPPET can simulate more processes than processors, that is, multiple processes can share individual processors, and are scheduled in a round-robin fashion. This allows one to evaluate various load-balancing schemes. Because PUPPET is loosely coupled, many schemes can be evaluated by running the program only once.

## 6.2 Future Work

There are several possibilities for extending the PUPPET project. One option is to bring features from other simulators to PUPPET. These include augmentation and/or instruction-level simulation, providing much higher accuracy than CPU time; an accurate network model for use when speed is less important or the network is simple (e.g., the Ethernet); the potential to have tight-coupling to simulate non-determinism; and simulation of an I/O subsystem.

Another possible direction is to add additional unique features. Dynamic process management is supported by the LAM MPI implementation, and it would likely be worth supporting in PUPPET; this would also have the consequence that a port to PVM (a popular message-passing system) would be quite simple. MPI-2 offers many new features and is currently under development; we hope to support it once it becomes official. Finally, we are currently considering the development of an MPI implementation that uses threads instead of UNIX processes to achieve high-speed task-switching for high-speed pseudo-parallelism.

# Appendix A

# Raw Data

This appendix contains the raw data that produced the plots of Chapter 5, except for the network-evaluator and simulation-speed results (for which there are hundreds of data points).

| | | 2 procs. | | 4 procs. | |
|---|---|---|---|---|---|
| Program | Problem Size | Act. | Sim. | Act. | Sim. |
| Merge-sort | 10,000 | 13.92 | 13.92 | 3.69 | 3.67 |
| | 20,000 | 55.69 | 55.50 | 14.06 | 14.25 |
| | 30,000 | 124.11 | 123.39 | 33.65 | 32.08 |
| Fan-in | 1,000 | 216.44 | 215.64 | 108.05 | 108.46 |
| | 2,000 | 1719.95 | 1706.43 | 843.06 | 823.50 |
| Fan-out | 1,000 | 223.34 | 220.73 | 119.47 | 112.51 |
| | 2,000 | 1773.13 | 1764.49 | 908.84 | 887.56 |

Table A.1: *Accuracy results for the Ethernet. The problem size represents the number of integers for the sorting programs, whereas it indicates the double precision matrix order for fan programs. Times are in seconds; act. and sim. stand for actual and simulated execution time, respectively.*

| | | 2 procs. | | 4 procs. | |
|---|---|---|---|---|---|
| Program | Problem Size | Act. | Sim. | Act. | Sim. |
| Fan-in | 1,000 | 6.66 | 7.38 | 3.62 | 5.32 |
| | 2,000 | 50.51 | 51.30 | 29.09 | 31.12 |
| Fan-out | 1,000 | 6.81 | 6.69 | 3.68 | 4.19 |
| | 2,000 | 54.58 | 51.76 | 40.05 | 28.00 |

Table A.2: *Accuracy results for Cholesky factorization on the SP2.*

| | | 1 proc. | | 2 procs. | | 4 procs. | |
|---|---|---|---|---|---|---|---|
| Program | Size | Act. | Sim. | Act. | Sim. | Act. | Sim. |
| GEMM | 800 | 7.01 | 7.00 | 3.69 | 3.79 | 1.91 | 2.22 |
| | 1600 | 55.99 | 55.70 | 28.70 | 28.87 | 14.70 | 15.30 |
| | 2400 | — | — | 96.06 | 99.32 | 47.83 | 51.20 |
| TRMM | 800 | 3.67 | 3.67 | 1.93 | 1.98 | 1.06 | 1.26 |
| | 1600 | 28.66 | 28.47 | 14.67 | 15.03 | 7.75 | 8.11 |
| | 2400 | 96.04 | 95.55 | 48.91 | 49.23 | 24.86 | 25.37 |
| SYRK | 800 | 3.72 | 3.71 | 1.97 | 2.05 | 1.11 | 1.35 |
| | 1600 | 28.85 | 28.65 | 14.86 | 15.16 | 7.91 | 8.36 |
| | 2400 | 96.30 | 95.84 | 49.25 | 50.10 | 25.63 | 26.39 |
| SYR2K | 800 | 7.41 | 7.39 | 3.92 | 4.06 | 2.22 | 2.59 |
| | 1600 | 57.62 | 57.29 | 29.67 | 30.25 | 15.83 | 16.52 |
| | 2400 | — | — | 98.24 | 103.02 | 51.35 | 55.37 |
| TRSM | 800 | 3.56 | 3.55 | 1.87 | 1.97 | 1.06 | 1.19 |
| | 1600 | 28.20 | 28.01 | 14.47 | 14.70 | 7.79 | 7.95 |
| | 2400 | 95.00 | 94.62 | 48.42 | 48.88 | 24.98 | 25.19 |

Table A.3: *Accuracy results for the scalable BLAS library on the SP2. GEMM, TRMM, SYRK, SYR2K, and TRSM stand for general matrix multiplication, triangular matrix multiplication, symmetric rank-K update, symmetric rank-2K update, and triangular solve, respectively. Dashes (—) denote problems that attempted to allocate data blocks larger than the maximum allowed, and so could not be run.*

| $P$ | Typical speedup | Square-root | Scaled speedup |
|---|---|---|---|
| 2 | 4.2071 | 2.0511 | 1.9766 |
| 4 | 15.1805 | 3.8962 | 3.6051 |
| 8 | 44.9873 | 6.7073 | 5.6018 |
| 16 | 68.5619 | 8.2802 | 5.8087 |
| 32 | 76.1486 | 8.7263 | 6.0604 |

(a)

| $P$ | Typical speedup | Square-root | Scaled speedup |
|---|---|---|---|
| 2 | 4.0408 | 2.0102 | 1.9793 |
| 4 | 17.7372 | 4.2116 | 3.8603 |
| 8 | 61.1046 | 7.8169 | 6.8294 |
| 16 | 146.5111 | 12.1042 | 9.0301 |
| 32 | 239.7313 | 15.4833 | 8.9786 |

(b)

| $P$ | Typical speedup | Square-root | Scaled speedup |
|---|---|---|---|
| 2 | 4.0744 | 2.0185 | 1.9969 |
| 4 | 17.3316 | 4.1631 | 3.9074 |
| 8 | 67.8781 | 8.2388 | 7.1970 |
| 16 | 201.0915 | 14.1807 | 11.1338 |
| 32 | 386.9224 | 19.6703 | 12.5176 |

(c)

| $P$ | Typical speedup | Square-root | Scaled speedup |
|---|---|---|---|
| 2 | 3.9459 | 1.9864 | 1.9976 |
| 4 | 15.9474 | 3.9934 | 3.9446 |
| 8 | 67.6620 | 8.2257 | 7.6259 |
| 16 | 245.9580 | 15.6830 | 13.2327 |
| 32 | 633.9893 | 25.1791 | 17.4460 |

(d)

Table A.4: *Various speedup metrics for the merge-sort program, as evaluated by the simulator, with a problem size of (a) 10,000, (b) 20,000, (c) 40,000, and (d) 80,000.*

| | Iteration number | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Attempt | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 23.63 | 21.94 | 21.24 | 20.64 | 20.37 | 20.33 | | | | | |
| 2 | 22.40 | 21.54 | 20.31 | 20.10 | 19.22 | | | | | | |
| 3 | 22.90 | 21.35 | 19.51 | 19.19 | 19.10 | 18.33 | 17.93 | 18.10 | 17.53 | 17.20 | 17.60 |

Table A.5: *Convergence of the simple load-balancing scheme for three attempts. Numbers are recorded until the last non-repetitive mapping is reached.*

| Case | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p9 |
|---|---|---|---|---|---|---|---|---|---|
| It. 0 | 5.98 | 69.33 | 86.67 | 52.03 | 12.50 | 25.25 | 94.35 | 36.96 | 12.36 |
| It. 9 | 59.38 | 60.01 | 71.81 | 70.90 | 60.11 | 56.51 | 74.12 | 66.48 | 74.77 |

Table A.6: *Utilization (in percent) of the nine processors used to solve a 32-process merge-sort problem with two process-to-processor mappings: the initial random mapping from attempt three and the best mapping found in attempt three.*

| | Number of processors | | | |
|---|---|---|---|---|
| Topology | 2 | 4 | 8 | 16 |
| Hypercube | 1.98 | 3.82 | 6.85 | 10.97 |
| Ring | 1.98 | 3.80 | 6.37 | 7.34 |

Table A.7: *Scaled speedup for the merge-sort program with two different topologies.*

# Bibliography

[1] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, Atlantic City, NJ, April 1967.

[3] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 23 of *Performance Evaluation Review*, pages 14–24, Ottawa, ON, May 1995. ACM Press.

[4] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of the 1993 Supercomputing Symposium Conference*, pages 588–597, Portland, OR, November 1993.

[5] Jürgen Brehm. PerPreT — A performance prediction tool for massively parallel systems. In *Quantitative Evaluation of Computing and Communication*

*Systems: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 977 of *Lecture Notes in Computer Science*, pages 284–298, Heidelberg, Germany, September 1995. Springer.

[6] Jürgen Brehm et al. A multiprocessor communication benchmark, user's guide and reference manual. Technical report, ESPRIT III Benchmarking Project, 1994.

[7] Eric Allen Brewer. Aspects of a parallel-architecture simulator. Technical Report MIT/LCS/TR-527, Massachusetts Institute of Technology, January 1992. A slight modification of Brewer's Master's thesis.

[8] Eric Allen Brewer and Chrysanthos N. Dellarocas. Proteus: A high-performance parallel-architecture simulator. *Performance Evaluation Review*, 20(1):247–248, June 1992.

[9] Eric Allen Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[10] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. Users' guide to `mpich`, a portable implementation of MPI. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1994.

[11] D. Brown, S. Hackstadt, A. Malony, and B. Mohr. Program analysis environments for parallel language systems: The TAU environment. In *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, pages 162–171, Townsend, TN, May 1994.

[12] John D. Bruner, Hoichi Cheong, Alexander Veidenbaum, and Pen-Chung Yew. Chief: A parallel simulation environment for parallel systems. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 568–575, Anaheim, CA, 1991. IEEE Computer Society Press.

[13] Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of the Supercomputing Symposium '94*, pages 379–386, Toronto, Canada, June 1994.

[14] Donald A. Calahan and David H. Bailey. Measurement and analysis of memory conflicts on vector multiprocessors. In Joanne L. Martin, editor, *Performance Evaluation of Supercomputers*, volume 4 of *Special Topics in Supercomputing*, pages 83–106. North-Holland, 1988.

[15] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for Level 3 BLAS. Technical Report TR-95-40, Department of Computer Sciences, University of Texas at Austin, Austin, TX, October 1995.

[16] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12 and UWCSE 93-06-06,

Sun Microsystems Laboratories, Incorporated, and the University of Washington, 1993.

[17] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 128–137, Nashville, TN, May 1994. ACM Press.

[18] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[19] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, June 1991.

[20] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 16 of *Performance Evaluation Review*, pages 4–11, Sante Fe, NM, May 1988. ACM Press.

[21] William D. Dally. Performance analysis of $k$-ary $n$-cube interconnection networks. *IEEE Transactions on Computers*, 39:775–785, June 1990.

[22] William D. Dally and Charles L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[23] Peter Davies, Philippe Lacroute, John Heinlein, and Mark Horowitz. Mable: A technique for efficient machine simulation. Technical Report CSL-TR-94-636, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, October 1994.

[24] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 99–107, Pleasant Run, IL, August 1991. CRC Press, Inc.

[25] Erik Demaine and Sampalli Srinivas. Routing algorithms on static interconnection networks: A classification scheme. *International Journal of Computer Systems Science and Engineering*, to appear.

[26] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. A distributed memory LAPSE: Parallel simulation of message-passing programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, volume 24 of *SIGSIM Newsletter*, pages 32–38, Edinburgh, Scotland, July 1991. IEEE Computer Society Press.

[27] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[28] James W. Dolter, P. Ramanathan, and Kang G. Shin. Performance analysis of virtual cut-through switching in HARTS: A hexagonal mesh multicomputer. *IEEE Transactions on Computers*, pages 669–680, June 1991.

[29] Jack J. Dongarra, Rolf Hempel, Anthony J.G. Hey, and David W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report ORNL/TM-12231, Engineering, Physics, and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 1993.

[30] Vincent Van Dongen, C. Bonello, and G. Gao. Data parallelism with High Performance C. In *Proceedings of the Supercomputing Symposium '94*, pages 128–135, Toronto, Canada, June 1994.

[31] Michel Dubois, Fayé A. Briggs, Indira Patil, and Meera Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 909–915, Penn State University, PA, August 1986. IEEE Computer Society Press.

[32] John B. Evans. *Structures of Discrete Event Simulation: An Introduction to the Engagement Strategy.* Ellis Horwood series in artificial intelligence. Ellis Horwood Limited, Halsted Press, a division of John Wiley & Sons, 1988.

[33] Sergio A. Felperin, Luis Gravano, Gustavo D. Pifarré, and Jorge L. C. Sanz. Routing techniques for massively parallel communication. *Proceedings of the IEEE*, 79(4):488–502, April 1991.

[34] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5, 1962.

[35] J. Gait. A probe effect in concurrent programs. *Software — Practice and Experience*, 16(3):225–233, March 1986.

[36] Patrick T. Gaughan and Sudhakar Yalamanchili. Adaptive routing protocols for hypercube interconnection networks. *Computer*, 26(5):12–23, May 1993.

[37] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderham. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[38] Alan George, Michael T. Heath, Joseph W.-H. Liu, and Esmond Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[39] Vasudha Govindan, Yoonho Park, Xida Li, Stacey Crear, and Olin Johnson. An overview of a MPI profiling environment for the NEC Cenju-3. In *Proceedings of the 2nd MPI Developers Conference*, pages 185–188, Notre Dame, IN, July 1996. IEEE Computer Society Press.

[40] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, 1994. For more information on MPI, see `http://www.mcs.anl.gov/mpi`.

[41] Dirk C. Grunwald and Daniel A. Reed. Networks for parallel processors: Measurements and prognostications. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, volume 1, pages 610–619, Pasadena, CA, January 1988. ACM Press.

[42] Vipul Gupta and Eugen Schenfeld. NetSim — A tool for modeling the performance of circuit switched multicomputer networks. In *Computer Performance Evaluation: Proceedings of the 7th International Conference on Modelling Techniques and Tools*, Lecture Notes in Computer Science, pages 180–192, Vienna, Austria, May 1994. Springer-Verlag.

[43] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, July 1988.

[44] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[45] Michael T. Heath and Jennifer A. Etheridge. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*, June 1994. Adapted from Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, May 1991.

[46] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, Argonne, IL, 1991.

[47] G. Hurteau, Vincent Van Dongen, and G. Gao. Overview of EPPP — an Environment for Portable Parallel Programming. In *Proceedings of the Supercomputing Symposium '94*, pages 119–127, Toronto, Canada, June 1994.

[48] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, September 1989.

[49] J. Robert Jump. *NETSIM: Reference Manual*, May 1993.

[50] P. Kermani and L. Kleinrock. A new computer communication switching technique. *Computer Networks*, 3:267–286, September 1979.

[51] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(1):25–42, February 1993.

[52] S. Madala. Concurrent C user's manual. Technical report, Dept. of Electrical and Computer Engineering, Rice University, Houston, TX, January 1987.

[53] I. Mathieson and R. Francis. A dynamic-trace-driven simulator for evaluating parallelism. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, volume 1 (Architecture), pages 158–166, Kailua-Kona, Hawaii, January 1988.

[54] Pankaj Mehra, Catherine H. Schulbach, and Jerry C. Yan. A comparison of two model-based performance-prediction techniques for message-passing parallel programs. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 181–190, Nashville, TN, May 1994. ACM Press.

[55] Message-Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–419, 1994.

[56] Alistair Moffat and Tadao Takaoka. An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$. *SIAM Journal on Computing*, 16(6):1023–1031, December 1987.

[57] Bernd W. Mohr, Allen D. Malony, and Kesavan Shanmugam. Speedy: An integrated performance extrapolation tool for pC++ programs. In *Quantitative Evaluation of Computing and Communication Systems: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 977 of *Lecture Notes in Computer Science*, pages 254–268, Heidelberg, Germany, September 1995. Springer.

[58] Martin A. W. Nemzow. *The Ethernet Management Guide: Keeping the Link*. McGraw-Hill, Inc., second edition, 1992.

[59] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–79, February 1993.

[60] Eddy Olk. PARSE: Simulation of message passing communication networks. In *Proceedings of the 27th Annual Simulation Symposium*, pages 115–124, La Jolla, CA, April 1994. IEEE Computer Society Press.

[61] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 1994.

[62] Daniel A. Reed and Dirk C. Grunwald. The performance of multicomputer interconnection networks. *Computer*, 20(6):63–73, June 1987.

[63] Eric Reiher, Herbert H. J. Hum, and Ajit Singh. Simulating networks of superscalar processors. In *Proceedings of the 27th Annual Simulation Symposium*, pages 125–133, La Jolla, CA, April 1994.

[64] Jennifer Rexford, James Dolter, Wu-Chang Feng, and Kang G. Shin. PP-MESS-SIM: A simulator for evaluating multicomputer interconnection networks. In *Proceedings of the 28th Annual Simulation Symposium*, pages 84–93, Pheonix, AZ, April 1995. ACM Press.

[65] David Samuelsson. System level interpretation of the SPARC V8 instruction set architecture. Technical Report R94:23, Swedish Institute of Computer Science, August 1994.

[66] David Kotz Saurab Nog. A performance comparison of TCP/IP and MPI on FDDI, Fast Ethernet, and Ethernet. Technical Report PCS-TR95-273, Dept. of Computer Science, Dartmouth Colledge, Hanover, New Hampshire, 1995.

[67] Matthias Schumann. Automatic performance prediction to support cross development of parallel programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 88–97, Philadelphia, PA, May 1996. ACM Press.

[68] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory par-

allel systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 171–180, Nashville, TN, May 1994. ACM Press.

[69] Michael D. Smith. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–59, Boston, MA, October 1992.

[70] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. PSIMUL — A system for parallel simulation the execution of parallel programs. Technical Report RC 11674 (#52414), IBM T.J. Watson Research Center, Yorktown Heights, NY, January 1986.

[71] Kimming So, Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. PSIMUL — A system for parallel simulation of parallel systems. In Joanne L. Martin, editor, *Performance Evaluation of Supercomputers*, volume 4 of *Special Topics in Supercomputing*, pages 187–211. North-Holland, 1988.

[72] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*, 1992.

[73] Leigh B. Stoller, Mark R. Swanson, and Ravindra Kuramkote. Paint: PA instruction set interpreter. Technical report, Dept. of Computer Science, University of Utah, Salt Lake City, UT, March 1996.

[74] Tadao Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43:195–199, September 1992.

[75] Manuel Mollar Villanueva (`mollar@inf.uji.es`), February 1996. Usenet posting on `comp.parallel.pvm`.

[76] Robert A. van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report TR-95-13, Department of Computer Sciences, University of Texas at Austin, Austin, TX, April 1995.

[77] Jack E. Veenstra and Robert J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, Durham, NC, January 1994.

[78] Herald Wabnig and Günter Haring. PAPS — The parallel program performance prediction toolset. In *Computer Performance Evaluation: Proceedings of the 7th International Conference on Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 284–304, Vienna, Austria, May 1994. Springer-Verlag.

[79] Herald Wabnig and Günter Haring. Performance prediction of parallel systems with scalable specifications — Methodology and case study. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 288–289, Nashville, TN, May 1994. ACM Press.

[80] Patrick H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, Oak Ridge, TN, September 1992.

[81] Deborra Zukowski, Manish Gupta, Madan Gopal, and Navin Budhiraja. XPOSÉ: A simulator for network development. In *Proceedings of the 27th Annual Simulation Symposium*, pages 59–69, La Jolla, CA, April 1994. IEEE Computer Society Press.